DESIGN AND EVALUATION OF A MULTISCALAR PROCESSOR

By
Scott Elliott Breach

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCES)

at the
UNIVERSITY OF WISCONSIN – MADISON
1998

Abstract

As the demand for processing power continues to escalate, future processor designs intended to meet this demand for performance must do so within the constraints of future implementation technology and the limits of practicable implementation costs. This thesis investigates a new type of processor based on the novel multiscalar paradigm. A multiscalar processor uses a "divide and conquer" strategy as a means to overcome the engineering challenges that face existing types of processors with respect to achieving high performance via improvements in instruction-level parallelism and clock speed.

This thesis focuses on the three most significant aspects of a multiscalar processor: instruction and data processing, instruction supply, and data supply. Detailed design descriptions and experimental evaluations are provided, identifying the impact of each aspect in terms of its individual performance as well as its contribution to overall performance. In addition, a comparison of realistic multiscalar and idealistic superscalar designs is provided to ascertain how this alternative approach performs relative to a well-known conventional approach.

The key components that dictate the characteristics of a multiscalar processor – processing units for instruction and data processing, hierarchical prediction and instruction memory for instruction supply, register file and data memory for data supply – are discussed in terms of the basic issues involved in their design. Moreover, the challenges/concerns for alternative designs are presented to focus on promising candidates for study. Each candidate is specified in terms of its overall structure and is evaluated under a range of design parameters to characterize its behavior and potential bottlenecks.

The performance comparison measures the speedup, relative to a baseline 1-wide out-of-order issue processor, of realistic multiscalar processors and idealistic superscalar processors. Given the microarchitecture and compiler capabilities assumed, this study indicates that even without an advantage in clock speed multiscalar processors can outperform superscalar processors, over a large range of configurations for the SPEC CFP95 programs, but over only a small range for the SPEC CINT95 programs. However, a key limitation of this work is that it is unable to factor in the clock speed difference between multiscalar and superscalar processors expected in actual implementations.

Acknowledgements

Putting all of the tangible pieces of this dissertation in perspective – the thousands of lines of simulator code, the hundreds of pages of thesis text and data, as well as the countless hours, days, weeks, even months of simulation time – it is clear that none can have the kind of lasting impact as the support and guidance of the comparatively modest group of people I would like to acknowledge here.

Above all others, I want to thank my family for their unconditional support throughout my entire life and during my (seemingly endless) educational endeavors. My mother, Karen, has always been there when I needed her, and I feel that she must take an equal share of the credit for my accomplishments. My grandparents, Ruth and Ken, have provided the stable, loving home to me and my mother that has allowed us to pursue this level of academic achievement. My cousin, Bill, has always kept me from forgetting exactly who I am as the years have gone by.

I also want to thank my wife, Pam. In the midst of this whole process, she won my heart, supported me (financially as well as emotionally), and put my needs above her own. With my own family far from Madison, her family welcomed me into their hearts and homes, and offered me their help and encouragement when I needed it. I cannot thank Pam's mother, Jeanette, enough for all of her home cooked meals and kindness.

The members of the Wisconsin Multiscalar group have provided an environment steeped in challenging interaction that has guided this work to its fruition.

To Manoj Franklin, we all owe thanks for sowing the seeds the rest of us cultivated. T.N. Vijaykumar has been my closest collaborator from the start to the finish of this work, battling the compiler while I did the same with the simulator. Andreas Moshovos has been an invaluable sounding board and has contributed key ideas to improve this work in particular and the Multiscalar project in general.

Todd Austin has offered his friendship since the day I arrived in Madison and continues to be a trusted confident, personally as well as professionally. Dionisios Pnevmatikatos graciously played the part of role model and mentor, though he probably never did and still does not realize it. Last but certainly not least, I would like to acknowledge the other members of the Multiscalar project for freely sharing their insights and ideas.

During my time in graduate school, I have had the pleasure of interacting with many more people than I can actually include in this space. I would like to express my appreciation for their friendship and influence on me as well.

I am especially indebted to the rest of the circle of friends with which I have shared the trials and tribulations of graduate life. Doug Burger, Babak Falsafi, Alain Kägi, and Subbarao Palacharla have helped me grow as a person in so many ways, that I cannot possibly account for them all here. I look forward to the honor of their friendship for the rest of my life. Steve Drucker and his family, though all the way back on East Coast, have always stood by me as a lifelong support group.

I am equally indebted to my advisor, Guri Sohi, who has contributed his invaluable experience as well as his effort and support in turning the multiscalar concepts into the reality of this thesis. The members of my preliminary and final defense committees, Jim Goodman, Mark Hill, Jim Smith, and David Wood, have offered their keen insight and judgment to improve this thesis and the research upon which it is based. I feel privileged to have had the opportunity to work in the company of these individuals who I honestly believe are second to none in the computer architecture community.

Finally, I would like to express sincere thanks to the Condor and Midship Projects, without whose generous assistance the vast quantity of simulation upon which this work is based would have been impossible. I would also like to acknowledge that this work was supported in part by a donation from the Intel Corporation and by NSF Grants CCR-9303030 and MIP-9505853, ONR Grant N00014-93-1-0465, and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

Scott E. Breach
University of Wisconsin – Madison, December 1998

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Efforts to improve the performance of serial programs continue to be the focus of much interest in academia and industry. General purpose microprocessors (often referred to as processors) are at the center of many efforts because machines designed for high performance on serial programs usually treat them as the principal component of the system. Clearly, one must not overlook the rest of the system as a critical factor in overall performance. However, the microprocessor is the component that sets the level of performance attainable, and other components are designed to support whatever level of performance it may attain.

Work to improve microprocessor performance continues to be synonymous with the design of uniprocessors. This strategy is the continuation of a well-established trend to enhance serial program performance by improving uniprocessor organization. Moreover, as argued in 1967 by Gene Amdahl (in his famous "Amdahl's Law" talk), a large powerful uniprocessor provides speedup on virtually every program [4]. This reasoning seems to hold as much truth for microprocessor designs of the future as it has for designs of the past. As a consequence, designing better microprocessors still remains the most direct way to influence the performance of serial programs and an important area for research.

1.1 Serial Programs

The reason serial programs receive such great attention from academia and industry is that these applications have been (and likely will continue to be) the dominant consumers of computing power. The importance ascribed to serial programs as computer applications, though, is not a coincidence. The serial program model is a simple and intuitive one that is assumed by most programmers when a program is written in a conventional high-level language (HLL), such as C, FORTRAN, *etc*. The semantics of the serial program model is that operations in the program are executed one at a time and in the order specified in the program. This view of the operation of a computer not only reflects the way people think, it describes the way computers actually functioned in their early incarnations. Over time, though, computers have developed into ever more complex machines that do not necessarily function in this simple way. Yet, the serial program model has allowed programmers to rely on this simple view when reasoning about their programs even though the computers on which their programs run have become complex.

Often, the impetus for increases in computer complexity is the need for increases in computer performance. The two most common methods to achieve higher computer performance are to perform individual operations faster or to perform more operations at the same time. The former approach usually relies on increases in the speed of the computer components, while the later usually relies on parallelism within the program run on the computer. Though it might seem that parallel execution of a serial program is not possible, there are often implicit opportunities for parallelism in such codes if the proper techniques are used to expose and to exploit them. Moreover, the well-known (to anyone who has tried) difficulties in terms of correctness and performance associated with explicit parallel programming often makes that approach an unattractive alternative. Thus, an implicit, as opposed to an explicit, approach to achieve parallel execution provides the key advantage that a programmer need not, in general, determine whether parallelism exists in a program nor consider how to express it.

This point is an important one because the parallelism within applications written as serial programs is usually either "hidden" from a programmer or is difficult for a programmer to reason about. Nonetheless, serial programs do contain parallelism [8, 13], sometimes significant amounts of it, though to tap this parallelism often requires the use of sophisticated hardware and/or software techniques [56]. In particular, aggressive microprocessors are designed to view a serial program as only the static specification of what operations are to be performed, not the dynamic realization of how the operations might actually be executed. The strength of the serial program model is that it allows the operation of the microprocessor to remain simple and clear enough that programmers can understand how to use it; yet, it does not preclude the use of (possibly radical) techniques to provide high performance parallel implementations, so long as the appearance of serial execution is maintained. The success of techniques such as pipelining, multiple instruction issue, instruction scheduling, and speculative execution (to name just a few) has helped the microprocessor assume a leading role in providing high performance on serial programs.

1.2 Microprocessor Performance

Over the course of the microprocessor's lifetime (the past 25 years), overall performance has grown exponentially [75]. If microprocessor performance is to sustain this rate of growth in the future, advancement in the techniques used to build microprocessors must continue. Techniques for improvements in microprocessor performance are driven in a complementary fashion by the forces of implementation technology and architectural innovation. As a result of shrinking feature sizes from fabrication advancements, implementation technology provides more transistors and faster clocks for the chips on which microprocessors are constructed. However, more transistors and faster clocks do not guarantee better performance. To take advantage of more transistors and faster clocks, architectural innovation in microprocessor designs must accompany changes in implementation technology. Furthermore, to provide the best performance, architectural innovations must avoid the weaknesses and exploit the strengths of the available implementation technology.

The use of architectural innovations to improve microprocessor performance is usually aimed at supporting the ability to execute instructions in parallel and/or in an order different from that specified in the original serial program. Regardless of what actually occurs within the microprocessor, it must appear that each instruction is executed one at a time in the original serial program order. Because individual instructions are the entities being executed in parallel, such techniques exploit what is referred to as *instruction-level parallelism* (sometimes ILP for short). However, performance is proportional to the product of ILP and clock speed. Thus, the most effective architectural techniques are those that propose ways to use the available transistors afforded by the implementation technology so as to maximize this product. That is, the goal of architectural techniques is to strike the best balance between ILP and clock speed for a given implementation technology.

1.3 Implementation Technology

Industry forecasters have generally been quite accurate in tracking improvements in implementation technology. The current semiconductor industry road map projects clock frequencies in the low gigahertz range within the next ten years [7]. This clock speed projection implies that future microprocessor must be able to sustain instruction completion rates in the range of four instructions per clock cycle in order to continue the trend in exponential performance growth [107]. Clearly, very aggressive microprocessor designs are required to achieve this degree of instruction-level parallelism.

Fortunately, it is forecasted that implementation technology will deliver an order of magnitude more transistors in the next ten years than are currently available along with the expected increase in their speed. Unfortunately, an important aspect of this future implementation technology is that logic delay scales well with feature size, but wire delay remains relatively constant. Wire delay has not been a concern in older implementation technologies because this delay has been a relatively small fraction of the clock cycle time. Now, in newer implementation technologies, wire delay is likely to become a larger fraction of the clock cycle time and hence may be more of a performance limiter [11].

The implications of these trends in future implementation technology are straightforward. In the past, on-chip communication was cheap, but transistors were too scarce to explore high degrees of instruction-level parallelism. In the future, transistors should be plentiful, but on-chip communication between these transistors could be expensive. This situation is encouraging with respect to microprocessor design, since the resources needed to enable high degrees of instruction-level parallelism appear to be forthcoming. However, it may be necessary to rethink the use of existing processing paradigms for the design of future microprocessors given the outlook in on-chip communication.

1.4 Design Directions

Present state of the art microprocessors, based on the superscalar paradigm, provide the ability to issue multiple instructions per cycle, often in an order different from that specified in the original program. Such processors use branch prediction techniques to fetch and decode a program's dynamic instruction sequence, building what is known as an *instruction window*. The individual instructions within this instruction window are analyzed for dependences and issued to functional units for execution based primarily on the availability of their operand data rather than on their order in the program. Upon completion, the instructions are buffered so that the processor state is only updated in the original program order (as dictated by the serial program model). This process, known as *dynamic instruction scheduling*, is what allows a superscalar processor to issue multiple, possibly out-of-order, instructions per cycle.

In practice, superscalar microprocessors (such as the Intel Pentium Pro), build an instruction window in the range of only a few tens of instructions and employ around four wide instruction issue to sustain the completion of somewhere between one and two (though often less than one) instructions per cycle on serial programs [10]. The modest degree of instruction-level parallelism of which existing microprocessor designs are capable is not sufficient to keep pace with the exponential performance growth expected from future microprocessor designs – instruction completion rates of four instructions per cycle. In particular, the existing designs of today do not appear to scale in terms of instruction-level parallelism or clock speed, given the expected trends in the implementation technology of tomorrow. Nonetheless, it is still advantageous to consider the directions for future designs based on what has been learned from designs of the past and the present.

It is widely acknowledged that to extract much instruction-level parallelism in serial programs it is often necessary to inspect a very large group of instructions from the dynamic instruction sequence for independent operations [8, 13, 17, 60, 98]. Furthermore, it is often necessary to apply to the resulting instruction window a peak issue rate that is far in excess of the average issue rate in order to sustain the highest degree of performance, due to the feast or famine nature of instruction-level parallelism in many serial programs [40]. These observations imply that, with respect to the present state of the art, future high performance microprocessors must be capable of building much larger instruction windows of potentially hundreds of instructions and of employing much wider issue capacity of ten or more instructions per cycle. Accordingly, to take advantage of such vast machine resources, future microprocessor designs must rely on far more aggressive techniques than those used at present to expose and to exploit instruction-level parallelism.

It is also widely acknowledged that the existence of control and data dependences in serial programs significantly constrains the ability of a microprocessor to seize opportunities for instruction-level parallel execution. Thus, the ability to speculate freely on both control and data dependences is expected to play a far more significant role in future microprocessor designs than it does in present designs. For control dependences, it may be necessary to predict many branches per cycle and to do so with great accuracy, so that a very large instruction window can be maintained. For data dependences, it may be necessary to access data in an extremely aggressive manner, using speculation wherever profitable, so that execution is serialized only where absolutely necessary. Clearly, to build microprocessors that are so much more ambitious than those that exist now will require designs that are well matched to their implementation technology.

A straightforward evolution of existing state of the art processor designs would simply involve the use of more aggressive control and data speculative execution as well as wider control and data paths to support the demands of larger processor structures. The drawback of this naive strategy is that it does not take into account that communication throughout the processor, including the number and the length of wires that provide it, is likely to grow dramatically as a result. In particular, aggressive speculation usually calls for higher bandwidth between the processing resources and the instruction supply and the data supply that feed them, further generating communication throughout the processor. In turn, the clock speed that sets the timing of these components applies ever more pressure to deliver values with lower latency to avoid constraining execution. Without a doubt, if existing designs are simply scaled, the characteristics of the resulting processor appear to be in direct opposition to the on-chip communication limitations expected for future implementation technology.

1.5 Conventional Processing Paradigms

A future microprocessor design must be able to meet the expected demand for performance in terms of instruction-level parallelism, yet do so within the constraints of future implementation technology and under the limits of practicable implementation costs. These concerns can be distilled into three questions concerning the use of conventional processing paradigms for future microprocessors designs. First, can conventional processing paradigms deliver the expected levels of instruction-level parallelism in a general purpose manner? Second, can conventional processing paradigms provide designs that fit well with future implementation technology? Third, can conventional processing paradigms foster implementations that reign in the escalating design and test costs of such sophisticated processors? A processing paradigm that can effectively address all of these concerns has the advantage of offering a combination of flexibility and high performance with low cost which usually plays a crucial role in a successful general purpose microprocessor implementation.

Another way of looking at these three concerns is as follows. Not only must the processing paradigm be capable of achieving high performance, it must be capable of achieving high performance on a wide range of applications without programmer intervention. Not only must the processing paradigm be applicable to an ideal design, its use must be profitable in an actual design that takes full advantage of the clock speed and transistor density afforded by the implementation technology. Not only must the processing paradigm support feasible implementations, it must provide an easy growth path from one design to the next to amortize the cost of (hardware and software) engineering development efforts over many processor generations. Yet, despite the diversity of conventional processing paradigms studied by academia and industry, only two conventional processing paradigms are being espoused by academia and industry for future microprocessor designs: the superscalar paradigm and the multiprocessing paradigm. Even so, it might be the case that novel paradigms, such as the one studied in this thesis, are ultimately the best fit for future microprocessors.

1.5.1 Superscalar Paradigm

The superscalar paradigm [2], in its conventional form, sequences through a serial program to dynamically create a window of instructions, schedules instructions from this window for execution, and communicates the results of executed instructions to other instructions waiting in the window [38]. The key to the superscalar paradigm is the characteristics of the window it uses to expose and to exploit parallelism. It can be difficult to establish a very large window, even with highly accurate modern control flow prediction. In particular, the need to predict each and every branch in a window of potentially hundreds of instructions seems to place a limit on just how large such a window might become. Even if the difficulties in exposing this window are overcome, the difficulties in exploiting even greater instruction-level parallelism within it are prohibitive. That is, because the instruction window is a single centralized structure, increases in its overall size and/or dimension must be supported via increases in wire length and/or number. This requirement does not fit well with future implementation technology since wire delay as much as transistor delay is expected to be the performance limiter. Moreover, it is difficult to amortize design and test costs from one generation of superscalar processor to the next since changing an old execution core with substantially different window and/or issue characteristics from those needed in the next generation usually necessitates a totally new development effort. The VLIW paradigm, a paradigm closely related to the superscalar paradigm, uses nearly the same approach subject to similar difficulties [15, 24, 71]. The major difference, though, is that since most of the details are handled at compile-time rather than run-time, dynamic problems appear in the form of their static counterparts.

1.5.2 Multiprocessing Paradigm

The multiprocessing paradigm [9], in its conventional form, relies on a compiler or a programmer to transform a serial program into a parallel one. This transformed program is usually composed of a number of control and data independent threads. The execution of the instructions of these threads is performed on multiple processors as independent dynamic windows of instructions. The communication of results among the executed instructions of the threads is accomplished via shared memory (with explicit synchronization) or message passing.

Given the vast number of transistors expected from future implementation technology, it appears to be possible to put a small multiprocessor on a single chip [62]; that is, it appears to be possible to construct a multiprocessor microprocessor. A multiprocessor of this design is a natural means to avoid centralized structures and is likely to have fairly low interprocessor communication delays. Moreover, it offers the advantage in terms of design and test of being able to reuse and to replicate many of the critical processor structures (e.g. the execution core) in moving from one generation to the next. However, it is still subject to the well-known limitations of automatic parallelization through compilation.

The present state of the art for this technology is able to find instruction-level parallelism implicit in some serial programs written in FORTRAN, but almost no serial programs written in C [3]. The root of the problem is that the conventional multiprocessing approach cannot in general provide the guarantees necessary to permit the correct execution of control and data dependent threads or the use of speculation across threads. Yet, these techniques are acknowledged as key aspects of any solution to find instruction-level parallelism in ordinary serial programs [46]. The multithreading paradigm, an approach closely related to the multiprocessing paradigm, makes use of the same basic mechanisms subject to similar limitations, but multiplexes the underlying hardware for better utilization.

1.6 Alternative Processing Paradigm

In the last few years, researchers have begun to consider extending the conventional superscalar [21, 65] and multiprocessing paradigms [63, 90] to combat some of the difficulties already described. To be sure, these efforts are important, albeit cautious, steps in the right direction. Nevertheless, the expectation of an unprecedented degree of instruction-level parallelism and the projection of a dramatic shift in design constraints may warrant more radical thinking in microprocessor design. This thesis investigates a novel approach for running serial programs called the multiscalar paradigm. The multiscalar paradigm first appeared about 1990 as a circa 2000 processing paradigm. It takes its name from its structure: multiple (super)scalar processing units that cooperate to execute a serial program in a parallel manner. The multiscalar paradigm provides an alternative to conventional processing paradigms that is specifically tailored to future implementation technology and readily accommodates a straightforward path from one processor generation to the next.

The multiscalar paradigm merges the strengths of the superscalar and the multiprocessing paradigms, yet relies on architectural innovation where needed to overcome their weaknesses. A multiscalar processor exposes a very large window of instructions by splitting the dynamic instruction sequence of a serial program – using hardware, software, or some combination thereof – into a collection of control and/or data dependent regions. It exploits instruction-level parallelism by executing these regions in parallel on multiple processing units via aggressive control and data speculation. Furthermore, it manages the serial semantics of control and data dependences implicit in the original program in a hierarchical fashion to support an efficient realization of the required mechanisms in an actual design. (A much more detailed description of multiscalar processors may be found in the next chapter.)

A multiscalar processor reaps instruction-level parallelism from the same source as a superscalar processor does. In both paradigms, the goal is to expose a vast window of instructions and to execute independent operations within that window. However, a multiscalar processor uses a “divide and conquer” strategy to avoid the engineering challenges faced by a superscalar processor. The multiscalar paradigm achieves the effect sought by the superscalar paradigm of a single very large window with very wide issue by using multiple smaller windows with narrower issue. It can realistically support high clock speeds because the behavior of key processor components is only proportional to small, narrow structures instead of large, wide structures. Likewise, it can realistically support high levels of instruction-level parallelism because the basis for window and issue scaling is adding more small, narrow structures (of the same design) rather than developing an even larger, wider structure.

A multiscalar processor is similar in overall structure to a multiprocessor. Like the multiprocessing paradigm, it provides the opportunity to reuse and to replicate critical processor structures in moving from one processor generation to the next so that design and test costs may be held in check. However, unlike the multiprocessing paradigm, the multiscalar paradigm supports threads that may be control and/or data dependent as well as control and data speculative execution of these threads. Because the dynamic behavior of serial programs cannot always be determined via static analyses, a multiscalar processor may have much greater flexibility to extract instruction-level parallelism than a conventional multiprocessor does, since concurrent execution need not be restricted to only those situations in which guarantees may be given about the control and data dependences in the program.

1.7 Multiscalar Processors

Much of the multiscalar work to date has been invested to introduce the fundamental ideas and mechanisms upon which the paradigm is based [25]. A comprehensive design and evaluation of the aspects of a multiscalar processor that are essential for high performance has been lacking. The foundation for such an investigation, though, has been laid in recent work that considers the roles of hardware and software for realistic implementations of multiscalar processors [87]. However, that work does not provide the level of detail or the amount of study needed to critique the multiscalar paradigm as a possible choice for future processor designs or to ascertain where efforts to improve performance ought to be invested. Moreover, there is a clear need to provide a performance comparison between the multiscalar paradigm and a conventional processing paradigm, such as superscalar, to provide a point of reference in terms of instruction-level parallelism and clock speed advantages that are needed to make a multiscalar design an attractive alternative to traditional designs.

This thesis explores in detail the microarchitectural aspects of one possible multiscalar processor organization. A companion thesis that explores in detail the compiler aspects of this same multiscalar processor organization complements this work [97]. Though multiscalar processor organizations other than the one considered by these interrelated works are possible, the total design space is simply too vast to study them all. As a result, much thought has gone into the organization used for these studies to ensure that it reflects realistic software/hardware choices/constraints that are comparable to those of an actual implementation. Moreover, the goal of these works is to further an understanding of the advantages and disadvantages of the multiscalar paradigm not to establish its ultimate realization (which will follow naturally if warranted). To conduct a methodical investigation, this thesis partitions the operation of a multiscalar processor into three fundamental aspects (devoting a chapter to each): instruction and data processing, instruction supply, and data supply.

In studying these aspects, the key components that dictate their respective characteristics – processing units for instruction and data processing, hierarchical prediction and instruction memory for instruction supply, register file and data memory for data supply – are discussed in terms of the basic issues that must be addressed in the design of each component. Furthermore, the challenges/concerns for alternative designs are presented to focus on promising candidates for study. Each candidate is specified in terms of its overall structure and is evaluated under a range of design parameters to characterize behavior and potential bottlenecks from the standpoint of its individual performance as well as its contribution to overall performance. In addition, a performance comparison of realistic multiscalar processors relative to idealistic superscalar processors is performed to give an impression of how this alternative approach might fair against a well-known conventional approach. Each chapter provides an overview of the aspect studied (in the beginning), a thorough description of its design and evaluation (in the middle), and a detailed summary (at the end).

1.8 Thesis Organization

The remaining chapters of this thesis are organized as follows. Chapter 2 reviews multiscalar processors and presents the particular organization studied in this thesis. Chapter 3 describes the experimental framework utilized throughout this thesis. Chapters 4, 5, and 6 delve into the fundamental aspects that comprise the operation of a multiscalar processor: instruction and data processing, instruction supply, and data supply. Chapter 7 offers a performance comparison between a range of conservative and aggressive realistic multiscalar configurations and idealistic superscalar configurations to gauge the contributions that instruction-level parallelism and clock speed must provide to make multiscalar processors an effective alternative to superscalar processors. Finally, Chapter 8 identifies the key contributions of this work, gives a summary of this thesis, and considers future directions for the study of multiscalar processors.

Chapter 2

Multiscalar Processors

This chapter describes the concepts that are the basis of multiscalar processors in general, as well as those that are the basis of the multiscalar processor studied in this thesis in particular. Section 2.1 postulates that multiscalar processors, though perhaps considered a revolution in processor architecture at the time of their inception, might instead be a natural evolution. Section 2.2 presents the execution model that multiscalar processors support. Section 2.3 identifies the principal design issues that need to be addressed for a multiscalar processor. Section 2.4 discusses the implementation details for the multiscalar processor studied in this thesis. Section 2.5 gives a summary of this chapter.

2.1 Architectural Evolution

According to the serial program model, a program executes by performing one instruction at a time driven by the control and data attributes specified in the program. Usually, a HLL program is compiled into a static binary that represents the computation specified by the program in terms of the base instructions of the machine on which it is to run. These base machine instructions, given by the instruction set architecture (ISA), are those usually assumed to execute according to the serial program model.

The success of modern processor designs may be attributed to the use of a progression of techniques, applied from one processor generation to the next in a cumulative fashion. These techniques aim to convert a serial, static specification into a dynamic, parallel execution that reduces the time to run a program. This work postulates that the course of processor architectural evolution which has gone from sequential to pipelined and pipelined to superscalar in the past, might proceed naturally from superscalar to multiscalar in the future as shown in Figure 2.1 and described below.

2.1.1 Sequential

A sequential processor is constructed to execute instructions more or less explicitly as described by the serial program model. The instructions of the program are performed in order, one after another. This program execution produces a dynamic instruction sequence corresponding to the path of program execution. The control and data dependences in the original program are adhered to implicitly, since each instruction executes in the order dictated by the serial program model. While this approach is attractive because of its simplicity, the time to execute the program is dictated by the total time to execute each individual instruction since only one is executed at a time.

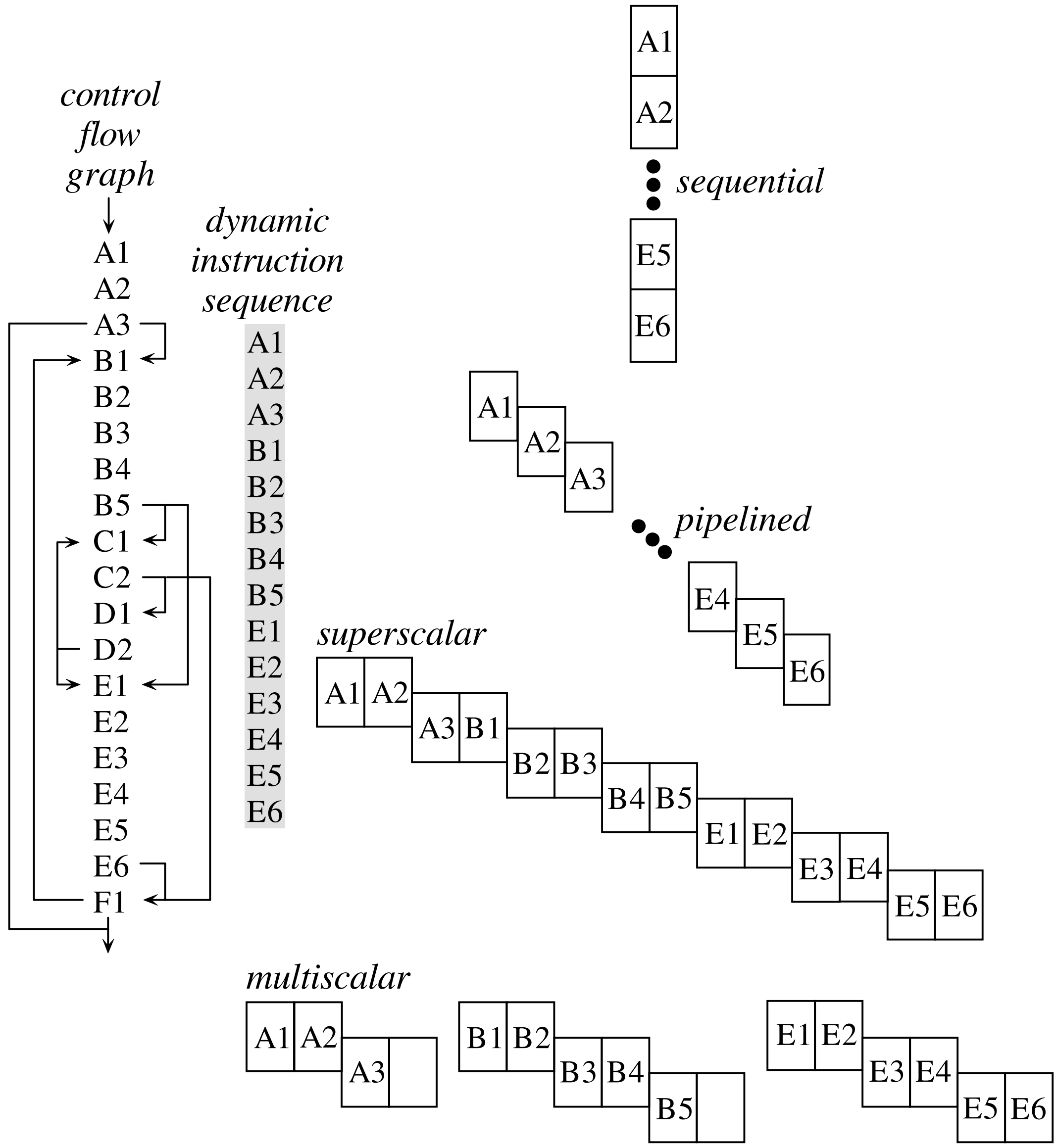


Figure 2.1: A postulated evolution of processor architecture.

2.1.2 Pipelined

A pipelined processor may improve upon the capabilities of a sequential processor by overlapping the execution of the individual instructions of the dynamic instruction sequence. This technique relies on the fact that instruction execution may usually be divided into a number of more primitive operations, such as fetch, decode, execute, *etc.* If these operations are independent, then the execution of instructions may be performed in a manner similar to an assembly line (or pipeline as it is called). While this approach is able to reduce the time to execute the program, since the time to execute individual instructions may be overlapped, it is still subject to the limitation that only one instruction is issued into the pipeline at a time.

2.1.3 Superscalar

A superscalar processor may improve upon the capabilities of a pipelined processor by issuing as well as overlapping the execution of multiple instructions at the same time. This technique provides additional machine resources and relies on the independence of instructions in the dynamic instruction sequence to support such concurrency. The success of this technique is tied to the ability to find independent instructions in the dynamic instruction sequence. For this reason, the most aggressive processors of this type look ahead into the dynamic instruction sequence to find such instructions. While this approach is able to reduce the time to execute a program, the engineering needed for extending it to look ahead farther and to find even more independent instructions for further reductions in execution time are forecasted to be a serious difficulty [107].

2.1.4 Multiscalar

A multiscalar processor may extend the capabilities of a superscalar processor by dividing the dynamic instruction sequence into multiple regions instead of treating it as a single region. A straightforward way of using this technique is to harness multiple superscalar processors, so that each executes a different region of the dynamic instruction sequence at the same time. In this way, each processor individually looks ahead only as far and finds only as many independent instructions as is practicable. The multiple processors collectively may provide the effect of looking ahead much farther and finding many more independent independent instructions than would be feasible for a single processor. Though unproven, the principles upon which this "divide and conquer" strategy are based seem to offer a means to further reduce execution time, yet overcome the most serious engineering limitations associated with superscalar techniques.

2.2 Execution Model

In order to appreciate the kinds of engineering challenges that are involved in building a multiscalar processor, it is helpful to understand how it performs its parallel execution of a serial program and what requirements serial semantics impose upon this process. Though a multiscalar processor may be composed of multiple (super)scalar processors, there are fundamental differences between the execution models of the two that warrant careful consideration, since these differences represent a significant factor in terms of the design of the processor and its performance.

2.2.1 Basic Idea

The basic idea is that a multiscalar processor divides the dynamic instruction sequence into a number of contiguous regions using hardware, software, or some combination of the two. Each of these regions is executed in parallel on a collection of processing units, each of which may be comparable to a conventional (super)scalar processor (or some other type of processor as appropriate). Because the regions of the dynamic instruction sequence are not necessarily control and/or data independent, and no guarantee is made to this effect, speculative execution is a key catalyst for this approach. The use of speculation requires the buffering of modifications made during the execution of each region in some form of speculative state to ensure that the architectural state of the processor may be preserved as needed in case of interrupts, exceptions, mis-speculations, *etc.* The speculative state modifications are performed on the architectural state (usually in program order for a straightforward design) at a later time when proper semantics are assured.

Figure 2.2 shows the control flow graph of a small code fragment with the dynamic instruction sequence corresponding to its execution. The dynamic instruction sequence has been divided into three regions, labeled **region 0**, **region 1**, and **region 2**. The goal of a multiscalar processor is to execute these regions in parallel as shown in the figure. If the instructions in each region are mostly independent of those in other regions, then parallel execution ought to occur in a straightforward manner. However, if the instructions in each region are somewhat dependent on those in other regions, then parallel execution might still occur, but it may be necessary to speculate and coordinate the execution of the regions to ensure correct semantic behavior. Regardless of what parallel execution transpires, each region is tied to the appearance of serial execution imposed by the dynamic instruction sequence. Since this requirement only imposes a partial order based on control and data dependences, not a total order, the actual execution order might be quite different from that assumed in the program.

2.2.2 Control Dependences

In order to supply correct instructions for execution, the control dependences within and among regions must be handled. The control dependences may be dealt with in a hierarchical manner. First, control dependences among regions may be handled to set up the parallel execution of the regions collectively. Second, the control dependences within regions may be dealt with individually to provide the instructions of each region.

In this approach, a global control flow prediction mechanism is used to step from one region of the dynamic instruction sequence to the next, with the implicit prediction of possibly

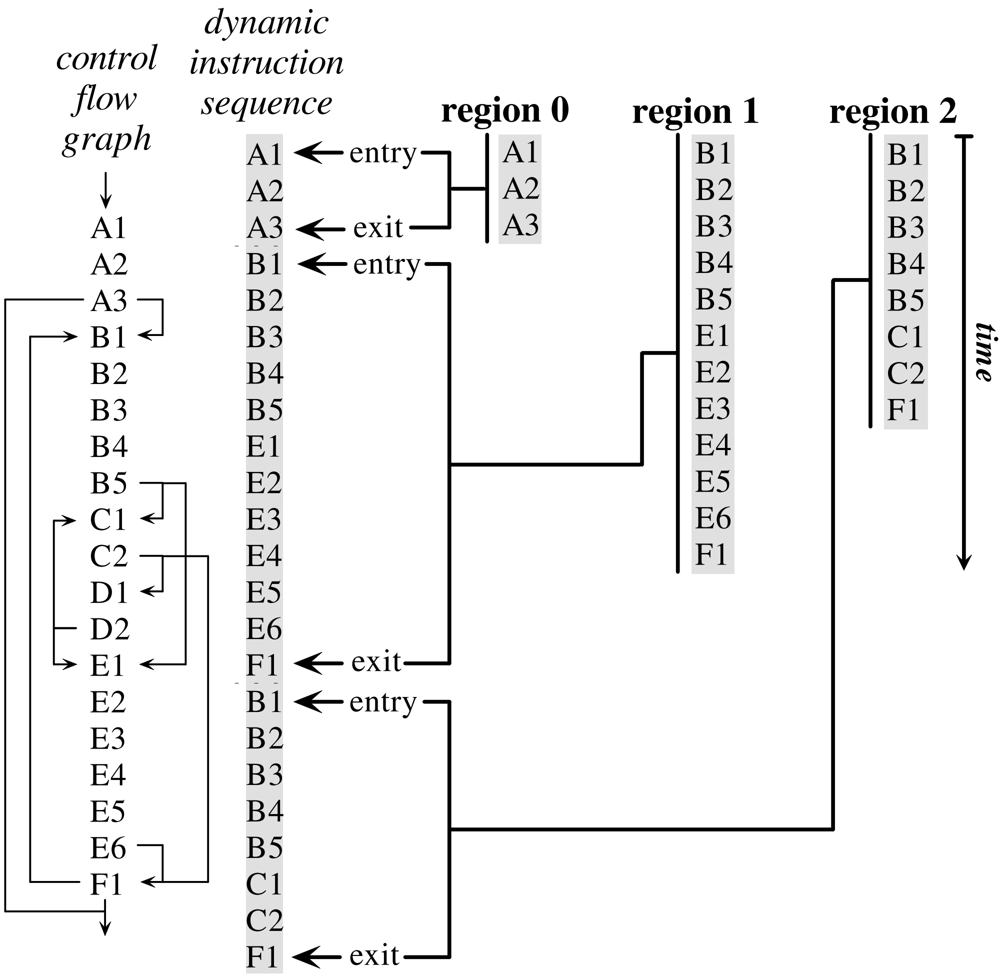


Figure 2.2: The idea of dividing the dynamic instruction sequence into regions and executing them in parallel that is basic to a multiscalar processor.

many control flow points within a region. Each global step assigns a region for execution on a processing unit by providing the program counter of the first instruction of the region.

On each processing unit a local control flow prediction mechanism is used to step through the region from one instruction to the next (as in a conventional processor) using this program counter. Each local step predicts the control flow points skipped over by the global step until the end of the region (*i.e.* the position of the global step) is reached. Among the processing units, multiple local steps occur in parallel.

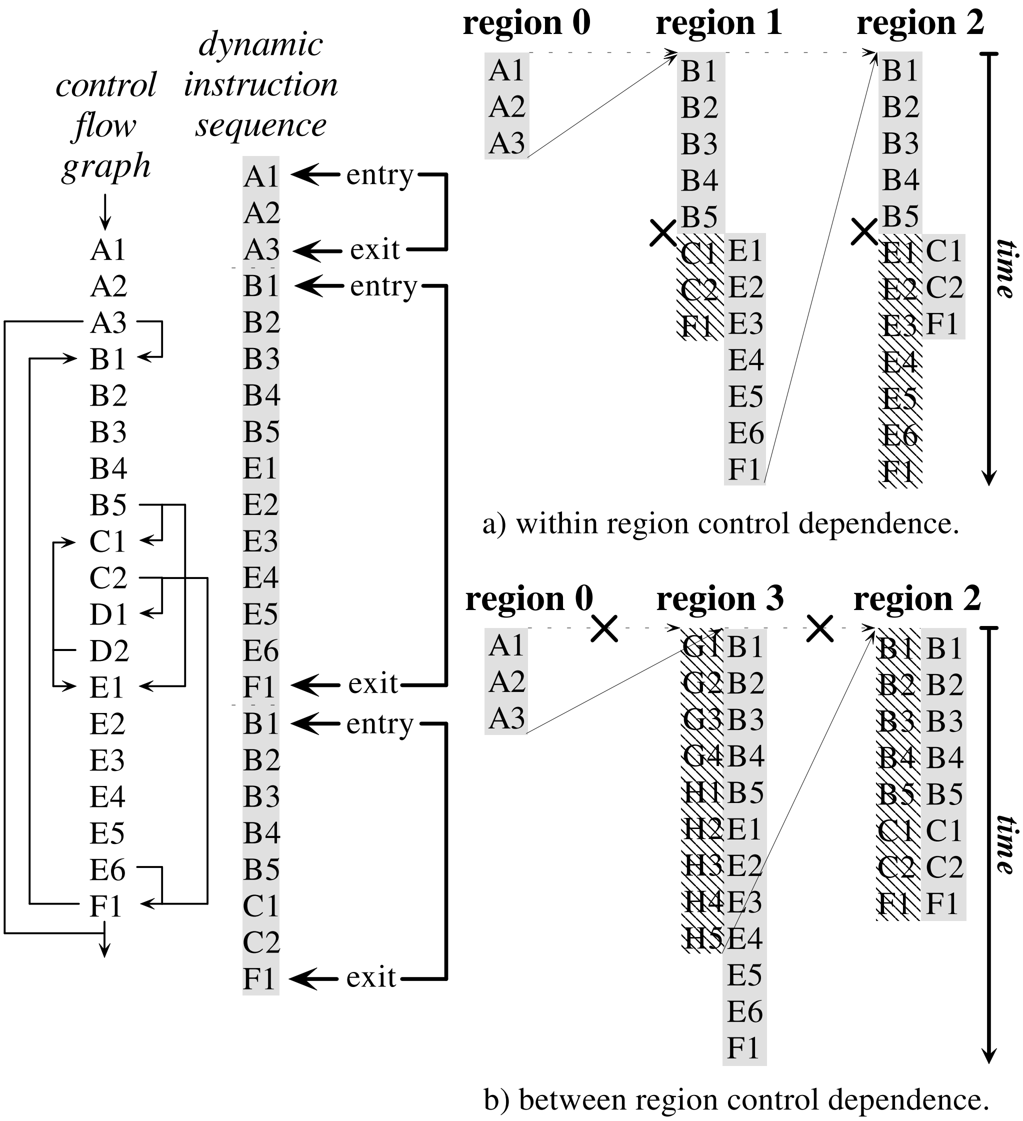


Figure 2.3: The handling of control dependences for a dynamic instruction sequence divided into regions.

Figure 2.3 shows the same small code fragment as before, but the focus is on the control dependences within and among the regions that execute in parallel. In part a) of the figure, the control flow between adjacent regions has been predicted from **region 0** to **region 1** and from **region 1** to **region 2**. In part b) of the figure, the control flow between adjacent regions has been predicted from **region 0** to **region 3** and from **region 3** to **region 2**. The dashed arrows, from the top of one region to the next, show the predicted control flow between regions. The solid arrows, from the bottom of one region to the top of the next, show the actual control flow. The control flow within regions is not indicated but may be identified by inspecting the dynamic instruction sequence and the control flow graph.

In part a) of the figure, the control flow between the regions is correct, but the control flow within some of the regions is incorrect, indicating a control dependence violation. At the time **region 1** and **region 2** are predicted, many control flow points within the regions (*e.g.* from **B5** to **C1** or **E1**) are predicted implicitly. As execution in the regions proceeds, prediction within each region may be performed explicitly. As shown in the figure, however, any control dependence violations within each region, such as **B5** to **C1** instead of **B5** to **E1** in **region 1**, need not affect the regions overall. Only the particular instructions within the region, as shown by the crossed-out shading, are affected, so long as the proper control flow from one region to the next is maintained.

In part b) of the figure, the control flow between the regions is incorrect. Though it was predicted that control flow would proceed from **region 0** to **region 3** to **region 2**, it actually proceeds from **region 0** to **region 1** to **region 2**. (**region 3** is some region of the dynamic instruction sequence corresponding to a part of the program control flow graph not shown in the figure.) As a result of the control dependence violation from **region 0** to **region 3**, both **region 3** and **region 2** are discarded, even though in the correct control flow sequence **region 2** executes in the same position. Though it may be possible to take advantage of such control independence, it requires additional knowledge about data independence that may make a successful implementation of any such mechanism problematic. A straightforward approach in which all execution following a control dependence violation is discarded, such as the one shown in the figure, is a viable solution so long as mispredictions are infrequent and/or the opportunities to salvage execution are few.

2.2.3 Data Dependences

In order to supply correct data for the execution of instructions, the data dependences within and among regions must be handled. As the execution within the regions proceeds, the instructions produce and consume values (held in register or memory storage) that must correspond to the order specified in the dynamic instruction sequence. The data dependences that dictate this order may be dealt with in a hierarchical manner (just as for control dependences).

If the producer and consumer instructions are within the same region, then the consumer must read the value for the location that the producer in its region writes, regardless of what other producers of the same location in other regions might write. Furthermore, if the producer and consumer instructions are in different regions, then the consumer must read the value for the location that the producer in the proper region writes.

Figure 2.4 shows the same small code fragment as before, but the focus is on the data

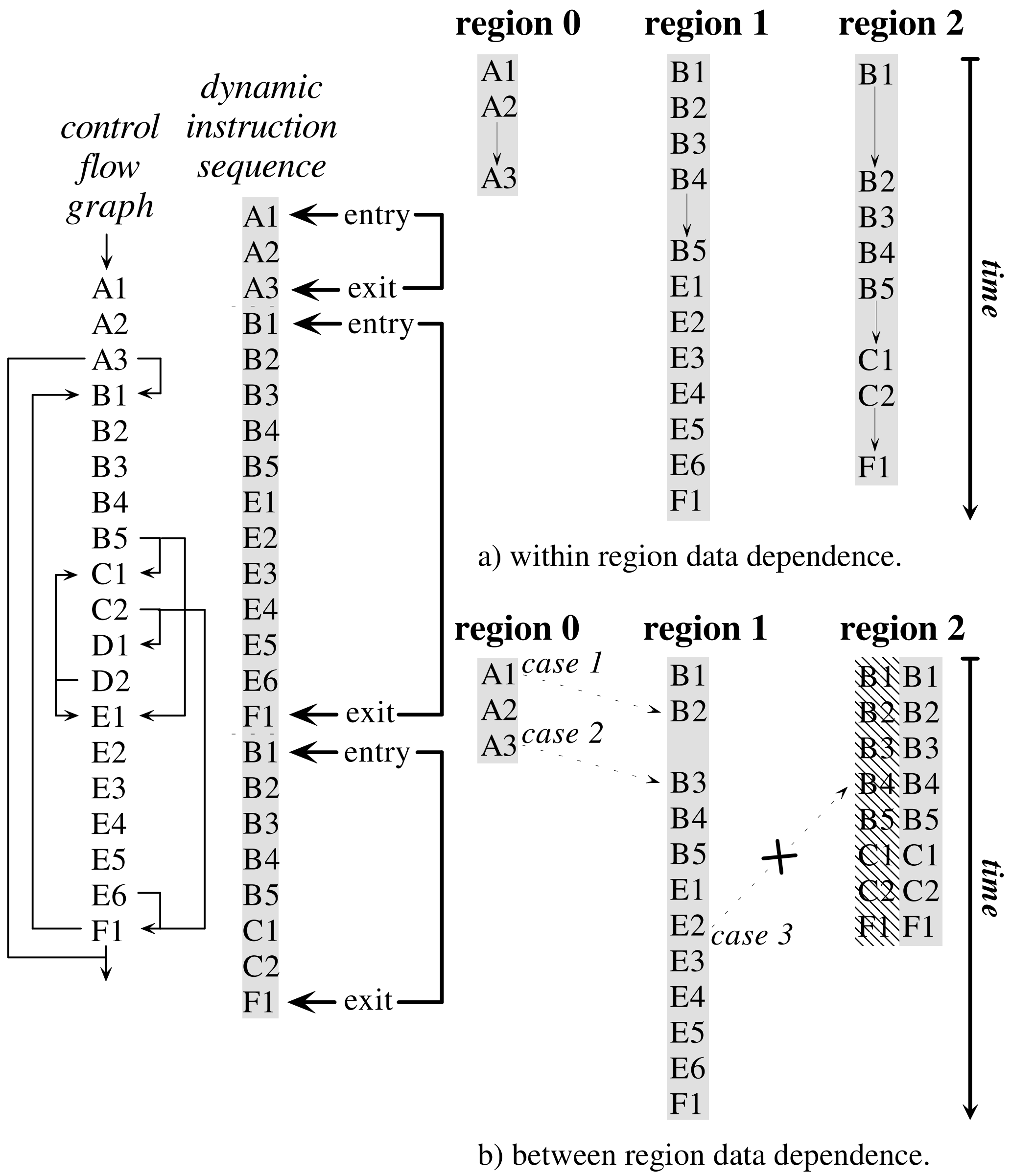


Figure 2.4: The handling of data dependences for a dynamic instruction sequence divided into regions.

dependences within and among the regions. The solid arrows between instructions identify data dependence within a region. The dashed arrows between instructions identify data dependence between adjacent regions. The length of an arrow inside of a region indicates the execution delay that is necessary to ensure the consumer instruction does not read until the producer instruction writes the value in question. There is no distinction made between data dependence for register or memory storage in this figure, since both must adhere to the same semantics. How this might be done differently for each type of storage will be discussed shortly.

In part a) of the figure, the data dependence among instructions within regions is always handled correctly. That is, a consumer instruction is always delayed until the value from a producer instruction is available. For example, if there is a dependence between instructions **B1** and **B2** in **region 2**, then the execution of **B2** (and in this example all instructions that follow) is delayed until the value from **B1** is available. This approach to resolve data dependence corresponds to what is done in most conventional processors. It is straightforward to always abide by data dependences in this way because each instruction in the dynamic instruction sequence is seen in the proper execution order, and the operands of each instruction may be analyzed to determine the relevant data dependences.

In part b) of the figure, three different cases for what might happen for the data dependences among regions are shown. In the first case, **B2** in **region 1** consumes data that **A1** in **region 0** produces, and execution occurs such that the producer and consumer instructions are performed in the proper order. As a result, no action is required to provide the correct semantics. If regions are determined and scheduled appropriately, the likelihood of this situation occurring may be quite high.

In the second case, **B3** in **region 1** consumes data that **A3** in **region 0** produces, but the order of execution would have resulted in a data dependence violation. However, no such violation occurs because the consumption of data by **B3** in **region 1** is synchronized with the production of data by **A3** in **region 0**, as indicated by the delay of the execution of **B3** in **region 1**. If data dependence can be determined either statically or dynamically, then data dependence violations in these instances may be avoided.

In the third case, **B4** in **region 2** does not consume the data that **E2** in **region 1** produces because the instructions do not execute in the proper order. As a result of this data dependence violation, all instructions in **region 2** as well as all instructions in any later regions are discarded. Other less drastic approaches, such as only discarding instructions that consumed incorrect data and those that are dependent upon them, may be possible, but how beneficial and/or viable these might be is uncertain for reasons similar to those given for control dependence violations.

2.3 Design Issues

The concepts upon which the multiscalar execution model is based are fairly straightforward. Yet, the keys to making it work well – how to build a multiscalar processor that maintains a fast clock and extracts a high degree of instruction-level parallelism – are choosing an effective design and supporting it with an efficient implementation. In particular, it is necessary to understand which aspects of the design ought to be handled by hardware or software, or perhaps some combination of the two.

This section identifies the principal design issues that need to be addressed in formulating a multiscalar processor design and tries to motivate to some extent the reasons for choosing between hardware and software design alternatives. The next section discusses the implementation details that go along with these design issues to provide a realistic specification for a multiscalar processor – in particular, the multiscalar processor that is studied in this thesis.

2.3.1 Region Selection

So far, the concept of dividing the dynamic instruction sequence into regions has been presented as an abstract process. An actual design, however, must provide concrete means of performing it. The process by which the dynamic instruction sequence is divided into regions may be performed via hardware, software, or some combination of the two. Clearly, the characteristics of the regions will have a significant effect on performance; though, it is beyond the scope of this work to provide a definitive answer to the question of which approach delivers the best performance. Nevertheless, two basic possibilities may be considered to appreciate the nature of such alternatives.

One possibility is that hardware would be given full responsibility to choose regions, dynamically selecting how the dynamic instruction sequence is divided and orchestrating its execution. Likewise, another possibility is that software would be given full responsibility to choose regions, statically selecting how the dynamic instruction sequence is divided and orchestrating its execution. Yet, just considering the possibilities without factoring in the implications of the choice is not wise. It is important to realize that such a design decision cannot be made arbitrarily because it may impact the interface between hardware and software (*i.e.*, the architecture) that the processor must support.

An interface in which region selection is implicit, transparent and only part of the implementation (hardware region selection), is attractive because it provides the ability to run existing binaries in unmodified form. However, the cost of this compatibility may be a loss of vision and flexibility in choosing the characteristics of the regions. On the other hand, an interface in which region selection is explicit, visible and part of the architecture (software region selection), is attractive because it exposes the performance potential of the multiscalar approach to the compiler. The compiler might create opportunities that would otherwise not exist or be out of reach of the hardware by restructuring the code, scheduling instructions within the code, and/or producing regions of vastly different sizes as needed.

The flexibility afforded by these capabilities in region selection may be required to ensure high performance on a wide range of applications. In particular, the grain of instruction-level parallelism as well as its control and data dependence characteristics can be quite different among applications (*e.g.*, numeric FORTRAN programs compared to non-numeric C programs). Not only might the compiler improve the overall performance in terms of instruction-level parallelism, it might improve the actual implementation by performing in software, analyses that are expensive or infeasible in hardware. Thus, the processor design need not be burdened with the complexity of performing dynamic analyses that are more easily performed as static analyses in the compiler. Instead, dynamic analyses can be reserved for performance critical situations in which static analyses are ineffective or impractical.

The ability of a multiscalar processor to select in a flexible manner the grain at which to extract instruction-level parallelism is likely to be a crucial advantage over conventional processors in terms of performance. However, if a multiscalar processor design relies only on hardware for region selection, then it may be constrained by the same difficulties that limit the ability of conventional processors to extract instruction-level parallelism. Thus, to ensure enough flexibility to explore the potential of a multiscalar processor, the process of region selection ought to involve the software, at least to some extent.

In keeping with these observations about the advantages and disadvantages of hardware versus software selection of tasks, the multiscalar processor studied in this thesis relies on the compiler to divide the dynamic instruction sequence into regions. The compiler partitions the control flow graph of the program into connected subgraphs, called *tasks*. The processor executes these subgraphs (or tasks) to generate the regions of the dynamic instruction sequence. As a result, the overall performance of the multiscalar processor studied in thesis is function of its compiler and it underlying design.

2.3.2 Task Hierarchy

A task corresponds to a contiguous region of the dynamic instruction sequence, that is generated by the execution of a connected subgraph in the program control flow graph. This connected subgraph is constrained to have a single entry point (otherwise performing any analyses is difficult), but may have multiple exit points. A task may correspond to part of a basic block, a basic block, multiple basic blocks, a single loop iteration, an entire loop, a function call, or various combinations thereof as specified by the compiler.

The execution of a task is bounded by an entry point, its first dynamic instruction, and an exit point, its last dynamic instruction. All instructions executed between the entry and exit points correspond to the region of the dynamic instruction sequence attributed to the task. Moreover, execution occurs as a sequence of tasks, so it always proceeds directly from one task to the next. (For the rest of this thesis, the regions of the dynamic instruction sequence as well as their corresponding connected subgraphs in the program control flow graph are referred to interchangeably as tasks.)

A key advantage of treating program execution as a sequence of tasks rather than just as a sequence of instructions is that it provides a means to decompose the difficult problem of dealing with dependences into two more straightforward problems which are less complicated to solve. That is, because a task encapsulates execution from the dynamic instruction sequence, it provides a natural means of coordinating dependences in a hierarchical manner. With the execution of the dynamic instruction sequence divided into tasks, dependences may be handled in a more or less decoupled manner. In particular, dependences within tasks (intra-task dependences) and dependences among tasks (inter-task dependences) may be handled by different means, as appropriate.

2.3.3 Intra-Task Dependences

The intra-task dependences involve handling the relationships among the instructions within a single flow of control in the dynamic instruction sequence, similar to a conventional processor. This problem is straightforward because each instruction executed in a task is seen in-order with respect to the dynamic instruction sequence. The relationships (or possible relationships) can be tracked and resolved as execution proceeds. Any of the well-known hardware and/or software techniques used for conventional processors may be appropriate to handle intra-task control and data dependences. The choice of techniques is flexible so long as the requirement that intra-task dependences adhere to serial semantics is met.

In most conventional processors, software scheduling techniques are used in conjunction with hardware scheduling techniques to improve performance. However, it is usually the case that hardware scheduling guarantees correctness in as much as software scheduling does not violate dependences. For the multiscalar processor studied in this thesis, intra-task dependences are handled by the processing units in nearly the same way as for conventional processors that use such a mix of static and dynamic scheduling. Though instructions may have been scheduled by the compiler, a processing unit predicts intra-task control dependences using a control flow prediction mechanism of some kind to speculate a window of instructions for the execution of a task. However, the effects of these speculated intra-task control dependences are encapsulated within a task. Moreover, their resolution, whether correctly or incorrectly speculated, is handled via the same mechanisms as for conventional processors.

A processing unit, however, does not speculate intra-task data dependences (likewise most conventional processors do not). Instead, the intra-task data dependences are synchronized for all instructions, though this synchronization is handled differently for register and memory storage. For register storage, dependences between instructions may be inspected as instructions are decoded because register locations are specified directly in the instructions. A dependent instruction may determine exactly which instructions produce register values it consumes. As a result, register data dependences delay instruction execution no longer than is actually necessary for serial semantics. For memory storage, the dependences between instructions usually may not be inspected because memory locations are specified indirectly via address calculations. A dependent load instruction may be unable to determine which store instructions produce memory values it consumes until all earlier addresses have been calculated. As a result, memory data dependences may delay instruction execution longer than is actually necessary for serial semantics.

2.3.4 Inter-Task Dependences

The inter-task dependences involve handling the relationships among the instructions within multiple flows of control in the dynamic instruction sequence. This problem is somewhat complicated because each instruction executed in a task may be seen out-of-order with respect to the dynamic instruction sequence, even though each task sees its instructions in-order. In particular, there may be instructions in earlier tasks which are not even known to exist (because sequencing may not reached that far) when a later task encounters its instructions. Moreover, even if all instructions were seen in-order, the distribution of the tasks among multiple processing units may make tracking and resolving relationships problematic.

This difficulty may be mitigated by using the property that tasks encapsulate execution in the dynamic instruction sequence. The dependences of an individual task might be aggregated to provide a dependence summary of the task. Using the dependence summaries from all earlier tasks, the execution of a later task may be guided with respect to honoring its dependences with the earlier tasks. If dependences in earlier tasks may be resolved exactly, then later tasks may use such information to avoid dependence violations. However, even if all dependences may not be resolved exactly, this information may be a valuable asset to assist in any speculation of these dependences. The specific nature of the dependence summaries depends on whether the software or the hardware provides them.

The hardware may provide more exact analyses, since it is based on actual execution, than may be possible using software. However, such hardware analyses may be expensive or impractical to perform and may not reflect the same depth of vision that may be available using software analyses. If the hardware provides the dependence summaries, then the processor gathers such information dynamically in a run-time managed structure. If the software provides the dependence summaries, then the compiler gathers such information statically in a compile-time managed structure which is in turn provided to the processor as part of the program. In either case, the dependence summary information needs to be maintained close to the processor for expeditious access during execution. For the multiscalar processor studied in this thesis, dependence summaries are provided by both the hardware and the software, though hardware analyses are used only where software analyses are not suitable.

The software provides the summary information based on the static analyses the compiler performs at compile-time. The static control dependence summary provides the number of task exits, the instruction types for task exits, and the program counter addresses that can be computed at compile-time for task exits. The static data dependence summary provides all registers that may be modified by the task, all registers of those that may be modified that are dead beyond the execution of the task, and the identity of the instructions which perform the last updates (the ones that satisfy dependences with other tasks) for the registers that may be modified but are not dead. The hardware provides summary information based on the dynamic analyses the processor performs at run-time. The dynamic control dependence summary provides the program counter addresses that could not be computed at compile-time for task exits. The dynamic data dependence summary provides the identity of instructions that might have a memory data dependence, based on their history of memory data dependence violations.

2.3.5 Dependence Speculation

Even with dependence summary information, it may not always be possible to avoid the need for dependence speculation nor the dependence violations that may result. The nature of the dependence analyses performed by the software and/or hardware gives rise to the following two cases. In the first case, the dependence summary information is exact. As a result, it may be used to eliminate the need for speculation and the possibility of dependence violations if the situation is amenable to such use. Nonetheless, dependence speculation may still be useful in this case in order to improve performance (*e.g.* value prediction [49, 50]). In the second case, the dependence summary information is not exact or perhaps is non-existent. As a result, it may not be used to ensure that no dependence violations occur, and the dependences involved must be handled by engaging in their speculation or waiting for their resolution.

2.3.5.1 Control Dependences

For control, the dependence summaries are exact (except for cases where a task exit addresses cannot be determined at compile-time), but this information alone does not assure that there is no need for dependence speculation. Since tasks follow only one of possibly many exits, just knowing all of the dependences is not enough to determine the next task (unless a task has only a single exit). However, it is impractical to wait for such control dependences to be resolved, as such an approach likely requires waiting until the last instruction of a task executes – a requirement that more or less eliminates the ability of the processor to expose instruction-level parallelism. Hence, the speculation of control dependences is performed as dictated by the situation.

2.3.5.2 Data Dependences

For register data, the dependence summaries are exact. The information provided by the compiler allows the production and consumption of register values to be synchronized. That is, no speculation of register data dependences occurs and no dependence violations result. The instructions in later tasks need wait for register values from earlier tasks only as long as it takes for the values to be produced and communicated. Under some circumstances, performance might be improved by speculating the register dependences [96], but this option is not explored further in this work.

For memory data, the dependence summaries are not exact. The compiler used in this work does not provide a memory analyses framework to handle such dependences in a general manner (for that matter, few compilers do). Without dependence summaries from the compiler, there are two clear options: wait for memory dependences to be resolved or speculate on memory dependences. The former approach avoids memory dependence violations, but significantly curtails chances to exploit instruction-level parallelism, which makes speculation the only tenable option. Another alternative, is the use of value prediction [49, 50] in conjunction with or in place of dependence speculation. However, this alternative is not studied in this thesis.

In practice, speculation of memory dependences usually results in few memory dependence violations during execution. The memory dependences that do cause violations (and sometimes considerable performance losses) are usually limited to a manageable set of all the memory dependences in the program. Moreover, this set of memory dependences exhibits the kind of temporal locality typical of most real reference streams [57]. As a consequence, these dependences may be dynamically tracked and resolved based on the history of mis-speculations to provide summary information for memory dependences. Though this information may not be exact, it may be used to predict cases where memory dependences (loads and stores) need to be synchronized, in a similar manner to register dependences.

2.3.5.3 Correct Speculation

The use of dependence speculation may lead to performance gains if it is correct. The results of speculative execution may be used in an optimistic manner throughout the processor to drive the extraction of instruction-level parallelism that otherwise might be unattainable. In general, handling dependence speculation when it is correct is straightforward because no dependence violations occur. Though the actual parallel execution order differs from the assumed serial execution order, correct behavior is assured so long as speculative updates are made to architectural state with the appearance of program order. Since tasks (just like the instructions within tasks) are encountered in order, it is appropriate for a multiscalar processor to buffer the speculative updates of tasks and perform them in program order at a later time – after all dependences have been resolved and all earlier tasks have performed their speculative updates – just as a conventional processor does for instructions. However, if dependence speculation is incorrect, leading to a dependence violation, then recovery action must be taken to provide correct behavior.

2.3.5.4 Incorrect Speculation

The cost of incorrect speculation depends on how often it occurs and the amount of work, usually incorrect but perhaps even correct, that must be discarded as a result. While the frequency of incorrect speculation may be comparable for a conventional processor and a multiscalar processor, the amount of work discarded may not be. If a conventional processor supports a much smaller window on the dynamic instruction sequence than a multiscalar processor, then it stands to reason that a multiscalar processor may discard a much larger amount of work than a conventional processor when an incorrect speculation occurs. Moreover, given that more instructions are discarded, it may be the case that more of them are correct rather an incorrect work. As a result, it may be worthwhile to salvage this work to reduce the cost of the incorrect speculation, depending on to what extent the method of recovery supports it.

Recovery may be handled in a number of different ways, each of varying complexity and potential benefit in terms of capacity to salvage correct work. The most complex and potentially beneficial approach is to discard only instructions in each task that have violated control and/or data dependences, and to reissue these instructions as well as those dependent upon them [25]. A slightly less complex and possibly beneficial alternative is to discard all instructions in tasks beyond the mis-speculation, but to rely on instruction reuse to reduce the amount of work that must be performed again [85]. However, if mis-speculations are an uncommon event, such complex mechanisms may be unwarranted, and a simple mechanism that discards all instructions in tasks beyond the mis-speculation may be sufficient. The multiscalar processor studied in this work uses this simple mechanism and focuses on providing mechanisms that try avoiding mis-speculation rather than on providing ones that rely on taking advantage of it.

2.4 Implementation Details

A multiscalar processor comprises many details concerning the software and hardware that support its implementation. This section discusses the implementation details that go along with these design issues to provide a realistic specification for a multiscalar processor – in particular, the multiscalar processor that is studied in this thesis. First, the program specification is discussed to explain what additional information is provided by the compiler (the software) to support multiscalar execution and how it is used to do so. Second, the microarchitecture (the hardware) of the multiscalar processor studied in this thesis is presented.

2.4.1 Program Specification

For the multiscalar processor implementation studied in this thesis, the compiler conveys static information to the processor to support the dynamic multiscalar execution model. That is, the program specification that the multiscalar processor is given contains more than just the instructions – usually all that is needed by a conventional processor. It contains additional information (described earlier) to support the multiscalar paradigm. This additional information allows execution to be treated not just as a sequence of instructions, but as a sequence of tasks, each of which in turn is a sequence of instructions. Both the compiler and the processor are used to support the multiscalar model because the combination yields an implementation that can taken advantage of the strengths of static as well as dynamic techniques according to the best fit for a particular situation. The details for the specification of this additional information is addressed in the text that follows.

Before providing the details, however, it might be helpful to reiterate the the actual nature of the information (described earlier). The program specification must indicate the entry and exit points for tasks to identify the subgraphs in the program control flow graph that correspond to the tasks selected by the compiler. In addition, it must provide the dependence summary information for which the compiler is responsible. The control dependence portion must provide the number of task exits, the instruction types for task exits to assist the predictors that provide their addresses, and the program counter addresses of the possible task exits that can be computed at compile-time. (The information for run-time computed addresses is provided by the components of the microarchitecture.) The data dependence portion must provide all registers that may be modified by the task, all registers of those that may be modified that are dead beyond the execution of the task, and the identity of the instructions which perform the last updates (the ones that satisfy dependences with other tasks) for the registers that may be modified but are not dead. (The information for memory is provided by the components of the microarchitecture.)

2.4.1.1 Binary Representation

There are a number of different means to provide a representation of the information described above in the program binary. The choice between these means is determined by what level of compatibility with existing program specifications is desired to be maintained. The major levels of compatibility, in order from more to less compatible, are (1) existing binary,

(2) existing instruction set architecture, and (3) clean slate with no existing constraints. The clean slate approach is the most straightforward, since all of the multiscalar specific program specifications can be factored into the instruction set architecture from its inception. Though this approach is acceptable for the purpose of research such as that performed in this thesis, it is often not an option in practice. As practical concerns are an important aspect of this work, providing more compatibility is deemed a necessary requirement.

A multiscalar binary may be generated from an existing binary by augmenting the binary with task entry and task exit information as well as the dependence summary information. It is probably necessary to use binary rewriting if this information needs to be added directly into the existing binary. Otherwise, all of this information might be incorporated into a different section of the binary text, placed off to the side of the original binary. The core of the binary, however, the fundamental instructions which describe the work the program may remain virtually the same. Only multiscalar specific instructions and any adjustments to relative addresses need to be accommodated. However, this approach does not allow the full exploration of compiler capabilities, since the binary rather than the source code provides the program description. Because a key aspect of this work is factoring in the contribution of the compiler to the performance of a multiscalar processor, this option does not seem to be well suited to the goals of this work.

A multiscalar binary may also be generated using an existing instruction set architecture. The task entry and task exit information as well as the dependence summary information provided by the compiler may be placed in a table and associated with each static instruction/task or incorporated into the program text as inlined information. The information, whether in a table or inlined, may be accessed along with the program text and concatenated to produce new instructions or directives as needed. These new instructions or directives may be maintained in the instruction cache, so that the overhead of concatenation occurs only in the case of a cache miss. Otherwise, the information may be kept in a separate cache, accessed in parallel with the instruction cache, to avoid any impact on the design of the instruction cache. If more fundamental instruction changes are required, these may be obtained by adding a few extra instructions to the base instruction set architecture or by overloading some of the existing instructions. This approach offers the flexibility needed to support the goals of this work. The implementation details of its use are presented next.

2.4.1.2 Illustrative Example

To clarify the actual means of representation for the additional information provided by the compiler, the following example is used to illustrate the implementation details of the program specification.

Consider the example HLL code, written in C, shown in Figure 2.5. In addition to the example HLL code, an assembly listing for a simple instruction set (similar to MIPS I but without delay slots of any kind) is shown as well in Figure 2.6. Each assembly instruction has been labeled with a letter (which basic block) followed by a number (which instruction in the basic block) to serve as a unique identifier within the assembly listing. The example code matches the control flow graph (CFG) as well as the number of instructions and number of basic blocks (A, B, C, D, E, and F) of the earlier examples in this chapter.

```
struct element {
    long symbol;
    struct element *next;
};

long *buffer;
long size;
long index;
long symbol;
struct element **list;
struct element *elem;

for (index = 0 ; index < size; index++) {
    /* get symbol for which to search. */
    symbol = buffer[index];

    /* do linear seach for symbol in list. */
    for (elem = *list; elem; elem = elem->next) {
        /* if symbol in list, done. */
        if (symbol == elem->symbol) {
            break;
        }
    }

    /* in symbol not in list; add to list. */
    if (! elem) {
        struct element *new;

        new = malloc(sizeof(struct element));
        new->symbol = symbol;
        new->next = *list;
        *list = new;
    }
}
```

Figure 2.5: Example C code.

```
# reg <-> program value
#
#  s0 <-> size
#  s1 <-> index
#  s2 <-> &buffer[index]
#  s3 <-> symbol
#  s4 <-> &list
#  s5 <-> elem
#  t0 <-> elem->symbol
#  t1 <-> list
#  a0 <-> argument to malloc
#  v0 <-> return from malloc

A1:     la      s2, &buffer[0]
A2:     li      s1, 0
A3:     bge     s1, s0, OUTFALL
OUT:
B1:     addu    s1, s1, 1
B2:     addu    s2, s2, 4
B3:     lw      s3, -4(s2)
B4:     lw      s5, 0(s4)
B5:     beq     s5, 0, INFALL
IN:
C1:     lw      t0, 0(s5)
C2:     beq     s3, t0, INBREAK
D1:     lw      s5, 4(s5)
D2:     bne     s5, 0, IN
INFALL:
E1:     move    a0, 8
E2:     jal     malloc
E3:     sw      s3, 0(v0)
E4:     lw      t1, 0(s4)
E5:     sw      t1, 4(v0)
E6:     sw      v0, 0(s4)
INBREAK:
F1:     blt     s1, s0, OUT
OUTFALL:
```

Figure 2.6: Example assembly code.

```
TASK0_DESC:
    exit = TASK1[bnt],TASK2[bt]
    create=s1,s2
    kill=
TASK0:
A1:   la     s2, &buffer[0]      F
A2:   li     s1, 0               F
A3:   bge    s1, s0, OUTFALL     EA
TASK1_DESC:
    exit=TASK1[bt],TASK2[bnt]
    create=s1,s2,s3,s5,a0,t0,t1,v0
    kill=a0,t0,t1,v0
TASK1:
OUT:
B1:   addu   s1, s1, 1           F
B2:   addu   s2, s2, 4           F
B3:   lw     s3, -4(s2)          F
B4:   lw     s5, 0(s4)
B5:   beq    s5, 0, INFALL
IN:
C1:   lw     t0, 0(s5)
C2:   beq    s3, t0, INBREAK
D1:   lw     s5, 4(s5)
D2:   bne    s5, 0, IN
INFALL:
E0:   release s5
E1:   move   a0, 8
E2:   jal    malloc
E3:   sw     s3, 0(v0)
E4:   lw     t1, 0(s4)
E5:   sw     t1, 4(v0)
E6:   sw     v0, 0(s4)
INBREAK:
F0:   release s5
F1:   blt    s1, s0, OUT         EA
TASK2_DESC:
    exit
    create
    kill
TASK2:
OUTFALL:
```

Figure 2.7: Example assembly code with tasks selected.

In this code segment, execution repeatedly takes a symbol from a buffer and runs down a linked list checking for a match of the symbol. If a match is found, there is no need to continue the search of the list. If no match is found, a new element of the list is allocated for this symbol. After an initial startup, additions to the list become infrequent, since most of the symbols match an element already in the list. Though this code might appear to be strictly serial, there is opportunity for parallel execution in a multiscalar processor if tasks are selected properly by the compiler.

2.4.1.3 Task Entry and Task Exits

The tasks specified within the static code produced by the compiler must indicate the boundaries of task execution to the hardware. The task boundaries are those instructions that represent an entry to or an exit from the task. In general, a task might have any number of entries, and a task might have any number of exits. However, in this implementation, a task has one entry and up to four exits.

Figure 2.7 shows the assembly code listing as it appears after the compiler has selected tasks for the example code segment. The compiler has partitioned the control flow graph into two tasks. The first task, **TASK0**, is the prologue of the outer loop that sets up the **index** and performs the entry test against **size**. The second task, **TASK1**, is an iteration of the entire outer loop, including all iterations of the inner loop as well as a possible function call to malloc. (A third task, **TASK2**, is shown as a place holder, but it is not part of the example code.)

The task exits are identified by associating an additional bit field with every instruction in the static code (using a table of such bits) that indicates the instruction (1) does not exit a task, (2) only exits a task only on the taken path, (3) only exits a task on the not taken path, or (4) always exits a task regardless of the path. (The taken and not taken specifications are for conditional branches.) In Figure 2.7, the exit of **TASK0**, instruction **A3**, has the additional field **EA** to indicate that its execution always exits the task, regardless of the path. Likewise, the exit of **TASK1**, instruction **F1**, has the same additional field.

The task entry might also be identified by using such an additional bit field. However, this information is not strictly necessary. The task exits must be identified explicitly in order to indicate that execution ought to stop. The task entry is identified implicitly by the instruction that starts the task, so no bit field needs to be associated with the instruction. As may be seen in Figure 2.7, no additional bit fields have been associated with the instruction that is the entry to **TASK0**, starting instruction **A1**, nor with the instruction that is the entry to **TASK1**, starting instruction **B1**.

2.4.1.4 Task Descriptor

Instead, each task entry has associated with it a task descriptor. A task descriptor is associated with a specific task by the fact that it immediately precedes the entry instruction of that task, identified in Figure 2.7 as **TASK0_DESC** for **TASK0** and as **TASK1_DESC** for **TASK1**. (Again, **TASK2_DESC** for **TASK2** is just a place holder.) This location provides direct access to the task descriptor using only the program counter corresponding to the task entry instruction (an address provided when a task is predicted). The task descriptor contains control and data dependence summary information needed to expedite the start of task execution.

The static control dependence information is utilized to facilitate prediction of the next task. (Any dynamic control dependence information cannot be provided as part of the binary.) For control dependences, the task descriptor indicates the number of task exits (indirectly). In addition, each task exit has associated with it a specifier to indicate the corresponding instruction type and an address (if available at compile-time) to identify the corresponding program counter value of the first instruction of the next task. (A task exit that is a jump and link instruction, provides the return address as well as the call address.) In Figure 2.7, the task descriptor for **TASK0** indicates that the task has two exits, **TASK1** via a not taken branch and **TASK2** via a taken branch; the task descriptor for **TASK1** indicates that the task has two exits, **TASK1** via a taken branch and **TASK2** via a not taken branch.

The static data dependence information for registers is utilized to set up the data dependences among tasks. (The dynamic data dependence information needed for memory cannot provided as part of the binary.) For data dependences, the task descriptor indicates all registers that may be modified by the task and all registers of those that may be modified that are dead beyond the task. The former is given as the **create** mask, and the later as the **kill** mask. Since a task may contain multiple basic blocks whose execution is governed by control dependences, it is not possible to determine statically which register values *will* be modified dynamically. As a result, the create mask given is conservative and includes all register values that *may* be modified. In Figure 2.7, the task descriptor for **TASK0** indicates that the task may modify registers **s1** and **s2**, but kills no registers; the task descriptor for **TASK1** indicates that the task may modify registers **s1**, **s2**, **s3**, **s5**, **a0**, **t0**, **t1**, and **v0**, but kills **a0**, **t0**, **t1**, and **v0**.

2.4.1.5 Task Control Dependences

The task control dependences are handled using the task descriptors and instruction extensions (described earlier), given in Figure 2.7, as follows. As execution starts at **TASK0**, the next task may be predicted using information in **TASK_DESC0**. The prediction mechanism uses the control dependence information to select between the two possible next tasks, **TASK1** or **TASK2**. Because the task descriptor contains the actual addresses of these next tasks, the prediction mechanism may access the exit address without seeing any of the instructions of **TASK0** (in particular without seeing the exit instruction, **A3**). **TASK0** executes until it reaches an instruction marked as a task exit, **A3**. At this point, the address to which the exit directs control and the address predicted for the next task are compared to determine whether a control dependence violation has occurred.

The situation is similar for **TASK1** using information in **TASK_DESC1**. The prediction mechanism uses the control dependence information to select between the two possible next tasks, **TASK1** or **TASK2**. Again, the task descriptor contains the actual addresses of these next tasks, so the prediction mechanism may access the exit address without seeing any of the instructions of **TASK1**. In contrast to the case for **TASK0**, the execution of the **TASK1** directs control across many branch instructions and may even contain a function call and its return. (The actual details of how the execution of the function call is handled are not provided here, but instead in the companion thesis discussing the compiler [97].) Nevertheless, once execution reaches an instruction marked as a task exit, **F1**, the actual and predicted addresses are compared to check for a dependence violation.

2.4.1.6 Task Data Dependences

The task data dependences for registers are handled using the task descriptors and instruction extensions (to be described later), given in Figure 2.7, as follows. As tasks are predicted, the create and kill masks in the task descriptor are used to determine for each task what registers may be modified by earlier tasks. A later task uses this information to place reservations on the indicated registers, so that if an instruction that accesses these registers executes, the instruction is made to wait for the value(s) to be communicated from the earlier tasks. For instance, when **TASK1** is selected to execute after **TASK0**, reservations are set on registers **s1** and **s2** for **TASK1**, because those registers are on the create mask, but not the kill mask, for **TASK0**. Thus, when instruction **B1** accesses **s1** or instruction **B2** accesses **s2**, the instructions are made to wait until the register values are communicated from **TASK0**. Likewise, further invocations of **TASK1** are made to wait until registers **s1**, **s2**, **s3**, and **s5** are communicated. However, the further invocations of **TASK1** do not wait for registers **a0**, **t0**, **t1**, and **v0** because these register are dead beyond each invocation and therefore not communicated to other tasks.

In accordance with the serial semantics of the dynamic instruction sequence, only the last update of a register in the task should be communicated to other tasks. The option exists to wait until all instructions in a task have been executed (*i.e.*, no further updates of registers are possible). However, this strategy is not expedient since it often implies that other tasks must wait, possibly a considerable period of time, for a value that is already available. In addition to knowing which registers a task may modify, the compiler also has knowledge of the last instruction in a task to update a register. The last updates of registers are identified by associating an additional bit field with every instruction in the static code (in the same way as for task exits) that indicates whether the instruction ought to forward to other tasks its modification of a register as the last update of the register. For instance, in **TASK0** instruction **A1** is the last update of register **s2**, and instruction **A2** is the last update of register **s1**, so each instruction has the additional field **F** to indicate it needs to forward a value to other tasks.

The situation is similar for **TASK1**, where instruction **B1** is the last update of register **s1**, instruction **B2** is the last update of register **s2**, and instruction **B3** is the last update of register **s3**. As a results, each of these instructions has the additional field **F** to indicate that each needs to forward their values to other tasks. However, the situation is somewhat different for register **s5**. This register is modified by instruction **B4**, but this instruction may not have performed the last update for the register, depending on the flow of control during the execution of the task. In particular, instruction **D1** of the inner loop may modify register **s5** many times until execution falls out of the loop. As a result, instruction **B4** may not be designated as the last update, but neither may instruction **D1** since it can execute many times. Though it does not occur in this code, the flow of control may even result in no instruction modifying a register. To deal with both of these situations, the compiler inserts **release** instructions to release the value for a register that can no longer be modified by the execution of the task. In **TASK1**, the compiler has inserted release instructions **E0** and **F0** to handle register **s5**.

2.4.2 Processor Microarchitecture

The chip technology in which multiscalar processors might be implemented is expected to provide many fast transistors on-chip, but on-chip communication between these transistor is expected be relatively slow [11]. In other words, wire delays are expected to dominate gate delays technologically speaking. To compensate for high relative communication to computation cost, the microarchitecture of a multiscalar processor must be structured such that the execution resources are divided (as is the program itself) in a way that maximizes local computation and minimizes global communication. The microarchitecture of the multiscalar processor that is the focus of this work is given in Figure 2.8. As shown in the figure, it has been partitioned into three fundamental functions: instruction and data processing, instruction supply, and data supply. Each of these functions is described briefly here, and studied in depth in the forthcoming chapters of this thesis.

Though there are many novel aspects in the multiscalar paradigm, the individual components of a multiscalar processor, as may be seen in the figure, are comparable to those in a conventional processor. As a result, it is certainly true that it may not be necessary to start from scratch in looking for solutions to multiscalar processor design problems. Nonetheless, it is by no means a foregone conclusion that adequate techniques exist. Often, the assumptions upon which a successful technique for a conventional processor is based are violated completely or stretched severely in the context of a multiscalar processor. In particular, the distributed nature of the processor microarchitecture usually represents a departure from the way most techniques are used in conventional processor designs. This basic difference is enough to warrant a careful re-evaluation of each of the major components that constitute the fundamental functions in a multiscalar processor.

2.4.2.1 Instruction and Data Processing

How instruction and data processing is performed in a multiscalar processor is studied in detail in Chapter 4. In a multiscalar processor, instruction and data processing is performed by a collection of processing units. A processing unit is comparable to the execution core of a conventional processor. It provides the execution resources (pipeline, functional units, reservations stations, reorder buffer, *etc.*) to perform the computation of a task. The multiple processing units of a multiscalar processor, though separate, act as a single, aggregate processor. This appearance is achieved by handling instruction and data processing in the hierarchical manner afforded by the use of tasks.

A processing unit executes instructions at a task granularity. A task is assigned to a processing unit for the lifetime of its execution. It executes all instructions on the assigned processing unit using only the resources available to that processing unit. The execution of tasks is speculative with respect to both the control and data relationships between instructions. As a result, each processing unit buffers the modifications made during task execution to speculative state and commits them to the architectural state when these control and data relationships have been resolved. So long as instructions within and among tasks appear to execute in the serial order, the semantics of the program are preserved.

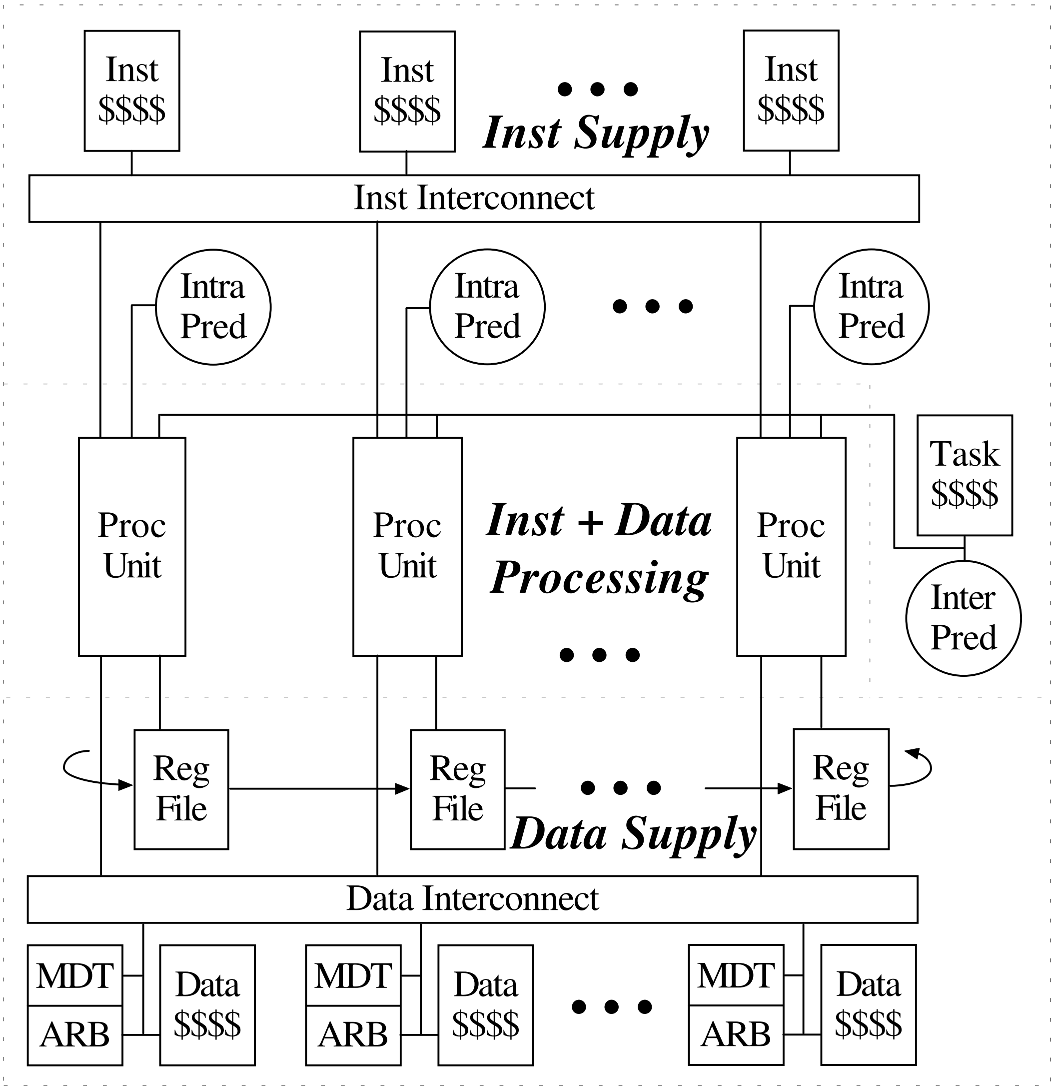


Figure 2.8: The multiscalar processor microarchitecture that is the focus in this thesis.

2.4.2.2 Instruction Supply

How instructions are supplied to a multiscalar processor is studied in detail in Chapter 5. In a multiscalar processor, instructions are supplied via a mechanism that performs what is known as sequencing. Sequencing is the process of bringing the dynamic instruction sequence of the program into the processor for execution. The proper handling of sequencing is significant for a processor that extracts instruction-level parallelism because it is the means by which the dynamic instruction sequence is exposed to the processor. The process of sequencing is usually driven by aggressive control flow prediction and expeditious instruction memory access. Control flow prediction speculates the dynamic path of program execution. Instruction memory access delivers the instructions (and perhaps other information, such as dependence summary information) for this dynamic path to the processor.

A multiscalar processor uses a two-level, hierarchical mechanism for instruction supply, with distinct intra-task and inter-task components. The inter-task component predicts the sequence of tasks for the dynamic path of the program. It consists of a single predictor with a task cache to provide access to the dependence summary information placed in the program by the compiler. The intra-task component predicts the sequence of instructions for each of the tasks sequenced by the inter-task component. It consists of multiple predictors, one for each processing unit, and a high bandwidth instruction cache to support the accesses of the processor units. The instruction cache may be organized as private caches, one at each processing unit, or as a bank of shared caches.

2.4.2.3 Data Supply

How data are supplied to a multiscalar processor is studied in detail in Chapter 6. In a multiscalar processor, data are supplied via a collection of storage locations addressed as separate register and memory namespaces (as in most modern so-called load-store instruction set architectures). The proper handling of both kinds of storage is significant for a processor that extracts instruction-level parallelism because the data dependences, for both register and memory storage, between instructions must be satisfied according to serial semantics regardless of what speculative and/or parallel execution may actually occur. Moreover, both register and memory storage mechanisms must provide high bandwidth and low latency access in order to supply data in a manner that does not constrain the ability of the processor to exploit instruction-level parallelism.

A multiscalar processor treats register and memory storage differently. The register storage consists of multiple register files, one for each processing unit, connected via a point-to-point unidirectional communication ring. The register file is organized as multiple private register files that are made to appear as a single shared register file. Each processing unit accesses its register file as a conventional processor does. However, communication between register files is orchestrated by the compiler and carried out by the processor to ensure that the register files remain consistent with respect to serial semantics. The memory storage consists of a high bandwidth data cache to support the access of the processing units. The data cache is organized as a bank of shared caches. Each bank, contains more than just a data cache, though. It contains an address resolution buffer (ARB) [27] and a memory dependence table

(MDT) [57] to sort out the memory data dependences and ensure serial semantics.

2.5 Summary

This chapter introduced the multiscalar concepts to provide a foundation for the rest of this thesis. In order to put this work in perspective, it was postulated that the lineage of processor architecture evolution that has proceeded in the past from sequential to pipelined and from pipelined to superscalar, might proceed in the future from superscalar to multiscalar. Given this prognostication, the execution model of multiscalar processors was presented. The basic idea of dividing the dynamic instruction sequence into regions, executing these regions in parallel on multiple processing units, and sorting out the control and data dependences among these regions in a hierarchical manner was described.

From these concepts were extracted the design issues of region selection, task hierarchy, intra-task and inter-task dependences, as well as dependence speculation that must be addressed in make a transition from the multiscalar execution model to a viable multiscalar processor. Further, the implementation details that correspond to answering the questions posed by the design issues were addressed to yield a realistic specification for the both the software and hardware aspects of a multiscalar processor – in particular, the multiscalar processor that is studied in this thesis. In addition, the principle microarchitectural features of this multiscalar processor designed were outlined in preparation for the rest of the thesis.

In the chapters that follow, a more detailed look is taken at the fundamental aspects of a multiscalar processor: instruction and data processing, instruction supply, and data supply. In these chapters, the components that comprise each of the aspects are described in detail and possible alternatives for each of the components are given. The strengths and weaknesses of each alternative are discussed, and where appropriate, simulation studies are performed to evaluate their effect on performance. However, before moving on to the design and evaluation that forms the body of this thesis, the experimental framework that is used to evaluate the various designs and their configurations is described in the next chapter.

Chapter 3

Experimental Framework

Before moving on to the study of multiscalar processors, this chapter describes the experimental framework in which their design and evaluation is performed. Section 3.1 provides a description of the methodology used to study multiscalar processors. Section 3.2 presents the benchmark programs considered in this thesis. Section 3.3 discusses the compiler that generates the code executed by the multiscalar processor studied in this thesis. Section 3.4 discusses the simulator that models the multiscalar processor studied in this thesis. Section 3.5 gives a summary of this chapter.

3.1 Methodology

The methodology used to study multiscalar processors provides an execution environment that is a combination of software and hardware. The software is a production-quality compiler that produces highly optimized code. The hardware is a simulator that models a wide range of processor designs as well as provides and gathers many statistics on behavior and performance.

3.1.1 Software

All C programs are compiled with a modified version of GNU GCC 2.7.2 targeted to an enhanced version of the MIPS instruction set architecture that does not include delay slots of any kind. All FORTRAN programs are compiled by first converting them to C with AT&T's F2C compiler. All programs are compiled with maximum optimization -O3, which performs classic optimizations as well as simple function inlining and loop unrolling. GNU GCC produces assembly files which are assembled with a version of GNU GAS ported to support a multiscalar enhanced assembly. The produced object files are compatible with the MIPS ECOFF object format. The programs are linked with GNU GLD. Standard library calls are implemented with a version of GNU GLIBC ported to support multiscalar instruction set enhancements and to emulate POSIX Unix system calls.

3.1.2 Hardware

All data for this work has been gathered from an execution-driven simulator that faithfully models the hardware organization of a multiscalar processor within a wide range of configurations. The simulator accepts annotated MIPS instruction set binaries produced by the compiler described above. It performs all of the operations of the processor and executes all of the program code, except system calls (which are handled as described above). All major aspects of a multiscalar processor are represented in the simulator in detail. (However, the virtual memory aspect is not modeled, as a proper, accurate consideration involves operating system details which have not been integrated into this infrastructure.) Of particular significance, bandwidth and latency for all components of the processor, whether due to interconnect or structure, are factored into the design and therefore reflected in the execution behavior.

3.2 Benchmark Programs

A natural choice for benchmark programs to study multiscalar processors is the SPEC CPU95 (or just SPEC95) suite [73]. SPEC95 is composed of two suites of benchmarks: SPEC CINT95 and SPEC CFP95. SPEC CINT95 is a set of eight compute-intensive integer benchmarks. SPEC CFP95 is a set of ten compute-intensive floating point benchmarks. These benchmarks are intended to provide a measure of compute-intensive (the 'C' in CINT95 and CFP95) performance of the processor, memory hierarchy, and compiler components of a computer system for comparison purposes. These benchmarks are not intended to stress the graphics, network, or I/O (aspects not covered by the methodology used in this work anyway). The SPEC95 benchmarks come with three different input sets: test, train, ref. The test input set is used to ensure that the SPEC95 tool set operates properly. The train input set is used to support feedback directed compiler optimization. The ref input set is used to perform the actual SPEC rating of a particular machine.

The ref input set is the one that ought to be used to measure performance in the experimental framework, since this input is used for performance ratings of actual machines. However, because this input is intended for real hardware, it runs for a relatively long time. The software used in this study to model the hardware is simply too slow to make this input a viable option for the extent of configurations studied in this work. As a result, the test and the train inputs are the more appropriate choices for simulated hardware, since these run for a much shorter time. Moreover, the train input is preferred over the test input because the benchmark behavior it produces is more representative of the benchmark behavior seen for the ref input than what the test input produces. A brief description of each of the benchmarks as well as the actual inputs used for this work are given below. In most cases, the train input is used for each benchmark. However, in a few cases, an exception is made if a much smaller input producing similar benchmark behavior is available from the test or ref set.

The SPEC CINT95 workload is the set of eight programs described in the text that follows as well as given in Table 3.1. 099.go is an internationally ranked go-playing program. 124.m88ksim is a chip simulator for the Motorola 88100 microprocessor. 126.gcc is based on the GNU C compiler version 2.5.3 from the Free Software Foundation. 129.compress is an in-memory version of the common UNIX compression utility. 130.li is the Xlisp interpreter for the lisp programming language. 132.ijpeg is an image compression/decompression utility for in-memory images based on the JPEG facilities. 134.perl is a stripped down version of an interpreter for the Perl language. 147.vortex is an object oriented database that is taken from a full object oriented database program called VORTEx. The train input set is used for all programs with the following exceptions. For 126.gcc, a smaller file from the ref input, jump.i, is used. For 132.ijpeg, a smaller file from the test input, specmun.ppm, is used. For 134.perl, only the scrabbl.in file from the train input is used.

SPEC CPU95 Integer Benchmark	Data Set	Description Command Line
099.go	train	Internationally ranked go playing program. go 50 9 < 2stone.in
124.m88ksim	train	Chip simulator for the Motorola 88100 microprocessor. m88ksim -c ctl.in
126.gcc	ref	GNU C compiler version 2.5.3. cc1 -quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -fschedule-insns2 -finline-functions -O jump.i -o jump.s
129.compress	train	In-memory version of UNIX compression utility. compress < test.in
130.li	train	Xlisp interpreter for lisp programming language. li train.lsp
132.ijpeg	test	In-memory image compression/decompression utility. ijpeg -image_file specmun.ppm -compression.quality 90 -compression.optimize_coding 0 -compression.smoothing_factor 90 -difference.image 1 -difference.x_stride 10 -difference.y_stride 10 -verbose 1 -GO.findoptcomp
134.perl	train	Stripped down interpreter for the Perl language. perl scrabbl.pl < scrabbl.in
147.vortex	train	Derivative of theVORTEx object oriented database. vortex vortex.in

Table 3.1: SPEC CINT95 workload.

The SPEC CFP95 workload is the set of ten programs described in the text that follows as well as given in Table 3.2. 101.tomcatv is a vectorized mesh generation program. 102.swim is a program that solves a system of shallow water equations using finite difference approximations. 103.su2cor is a Monte-Carlo method applied to the computation of masses of elementary particles in the framework of the Quark-Gluon theory. 104.hydro2d is a solver of Hydrodynamical Navier-Stokes equations to compute galactical jets. 107.mgrid is a multi-grid solver in a 3D potential field. 110.applu is a solver of parabolic/elliptic partial differential equations. 125.turb3d is a program that simulates isotropic, homogeneous turbulence in a cube, performing a large 1D FFT. 141.apsi is a solver of problems regarding temperature, wind, velocity, and distribution of pollutants for weather prediction. 145.fpppp is a solver for the Gaussian series of quantum chemistry (two electron integral derivative). 146.wave5 is a program that solves Maxwell's equations and particle equations of motion on a Cartesian mesh under a variety of field and particle boundary conditions. The train input set is used for all programs, except for 103.su2cor, 104.hydro2d, and 107.mgrid. For these benchmarks, the test input is used.

SPEC CPU95 Floating Point Benchmark	Data Set	Description	Command Line
101.tomcatv	train	Vectorized mesh generation program.	tomcatv < tomcatv.in
102.swim	train	System of shallow water equations.	swim < swim.in
103.su2cor	test	Monte-Carlo method applied to elementary paticles.	su2cor < su2cor.in
104.hydro2d	test	Hydrodynamical Navier-Stokes equations.	hydro2d < hydro2d.in
107.mgrid	test	Multi-grid 3D potential field.	mgrid < mgrid.in
110.applu	train	Parabolic/elliptic partial differential equations.	applu < applu.in
125.turb3d	train	Isotropic, homogeneous turbulence in a cube.	turb3d < turb3d.in
141.apsi	train	Weather prediction calculations.	apsi
145.fpppp	train	Gaussian series of quantum chemistry.	fpppp < natoms.in
146.wave5	train	Maxwell's equations and particle equations of motion.	wave5 < wave5.in

Table 3.2: SPEC CFP95 workload.

3.3 Compiler Aspects

Knowing the characteristics of the software for which the hardware is designed is the crucial factor in appreciating the components evaluated in the chapters to come. The performance and behavior of the multiscalar processor studied in this thesis are tied directly to the tasks produced by the compiler that execute upon it. As the chief determinant of the nature of instruction-level parallel execution, it is important to know the characteristics of the tasks in order to understand the performance wins and losses in a multiscalar processor. As might be expected, the actual algorithms and heuristics used to determine the tasks can be quite involved. Rather than discuss the means by which the tasks are determined (since it is beyond the scope of this work), it is preferable to describe the resulting tasks from qualitative and quantitative viewpoints. A detailed treatment of this subject may be found in a companion thesis to this one that studies compiling for the multiscalar paradigm [97].

3.3.1 Qualitative

The tasks for the integer benchmarks are influenced primarily by heuristics to leverage data dependence in task selection. Only register data dependences and simple memory data dependences (global scalars and stack variables) are considered since comprehensive memory data dependence analysis is not performed by the compiler. The most common form of task produced by this strategy is one whose entry point has few register or simple memory data dependences with respect to other tasks and whose constituent instructions have mostly register and simple memory data dependences among one another. These tasks have few basic blocks due to the density of control and data dependences in the integer codes. Moreover, the basic blocks are relatively small, usually containing only a few instructions.

The tasks for the floating point benchmarks are influenced primarily by heuristics to leverage control dependence in task selection. In particular, loop bodies are identified with the expectation that running them in parallel may support concurrent execution. The most common form of task produced by this strategy is one whose entry point is the head of a loop body and whose constituent instructions are the loop body itself. The existence of register and memory data dependences do not constitute as significant a factor in task selection as for the integer benchmarks. These tasks have one to many basic blocks due to the well-behaved nature of control and data dependences in the floating point codes. Moreover, the basic blocks are relatively large, often containing many instructions.

3.3.2 Quantitative

There are two major quantitative aspects of tasks that are of particular significance.

The first aspect is the relative dynamic instruction counts for scalar compiled code and multiscalar compiled code. The reason this aspect is a factor at all is because the compiler performs code structuring and scheduling which educes performance on a multiscalar processor but which produces no useful effect on a scalar processor. In fact, these compiler transformations have the effect of increasing the dynamic instruction count. The resulting dynamic instruction counts for scalar and multiscalar compilations as well as the change are shown in Tables 3.3 and 3.4 for the integer and floating point benchmarks, respectively. The arithmetic mean of the change is reported at the bottom for each table. Because the integer code is divided into tasks that are much smaller than those of the floating point code (as described below), this overhead is more than twice as much for the SPEC CINT95 (around 5%)

SPEC CPU95 Integer Benchmark	Dynamic Scalar Inst Count	Dynamic Multiscalar Inst Count	Change
099.go	553.8M	589.1M	+ 6.4%
124.m88ksim	121.7M	131.2M	+ 7.9%
126.gcc	172.6M	181.9M	+ 5.4%
129.compress	36.4M	38.5M	+ 5.7%
130.li	182.9M	200.4M	+ 9.6%
132.ijpeg	553.4M	594.6M	+ 7.4%
134.perl	39.3M	41.7M	+ 6.1%
147.vortex	2820.3M	2958.8M	+ 4.9%
			+ 5.3%

Table 3.3: Dynamic instruction counts for SPEC CINT95 benchmarks.

SPEC CPU95 Floating Point Benchmark	Dynamic Scalar Inst Count	Dynamic Multiscalar Inst Count	Change
101.tomcatv	13855.9M	14036.4M	+ 1.3%
102.swim	753.1M	765.0M	+ 1.6%
103.su2cor	1099.9M	1114.0M	+ 1.3%
104.hydro2d	1130.1M	1164.8M	+ 3.1%
107.mgrid	5568.4M	5905.4M	+ 6.1%
110.applu	649.3M	660.0M	+ 1.6%
125.turb3d	19621.2M	20034.8M	+ 2.1%
141.apsi	2846.5M	2910.2M	+ 2.2%
145.fpppp	442.2M	446.1M	+ 0.9%
146.wave5	3884.8M	3973.8M	+ 2.3%
			+ 2.3%

Table 3.4: Dynamic instruction counts for SPEC CFP95 benchmarks.

as for the SPEC CFP95 (around 2%).

The second aspect is the dynamic instruction counts of the tasks themselves. As shown in Tables 3.5 and 3.6 for the integer and floating point benchmarks, respectively, it is clear that integer tasks are small and floating point tasks are large, in relative terms; though, in both cases, the standard deviation of the task size is quite significant. In fact, it can be as much as and even many times the task size, indicating that relatively small and relatively large tasks (compared to the average) may execute at the same time. The arithmetic mean for the average task size and the standard deviation is reported at the bottom for each table. As a result of this disparity a natural question arises: what is the "correct" task size? Unfortunately, there is no definitive black or white answer to this question. As with most real problems, the truth of the solution lies in the gray that is somewhere in between. Nevertheless, there are some useful observations with respect to the bottom line on task size that may provide insight into this quandary.

3.3.3 Bottom Line

If tasks are too small, the instruction window may be too limited to support much instruction-level parallelism. This situation is aggravated by the fact that it is difficult to amortize the overhead associated with a task if it represents a relatively limited amount of actual execution. If tasks are too large, the cost of incorrect speculation may be too high to warrant the parallel execution of such tasks. This situation is aggravated by the fact that speculation itself can be extremely expensive (even if correct) as the cost to buffer all execution may be prohibitive. In reality, determining the correct task size is a balancing act between these extremes.

The most important factor in this balancing act is not the task size, per se. Instead, the task size must reflect the grain of instruction-level parallelism dictated by the particular application or particular parts of the application. The empirical evidence obtained in work on the multiscalar compiler [97] seems to suggest that the best performance for integer programs may be found with the small grain afforded by including dependence chains for registers as well as memory global scalars and stack variables within tasks. Whereas, the best performance for floating point programs may be found with the large grain afforded by enclosing the entire loop bodies within tasks.

3.4 Simulator Aspects

The two main ways to investigate machine organizations are to build them or to simulate them. The former approach is referred to as hardware prototyping, while the later is referred to as simulation. In general, the two approaches have complementary advantages and disadvantages. A hardware prototype is accurate and fast because it is real hardware, but as a result is expensive to construct and fairly inflexible. A simulator is relatively cheap to construct and very flexible, but because it is written in software, is slow and may be inaccurate. (Speed and accuracy are often inversely proportional.) Given the cost and time involved in building today's processors, a computer architect is forced to develop simulators that are both fast

SPEC CPU95 Integer Benchmark	Dynamic Inst Count	Dynamic Task Count	Average Task Size	Standard Deviation Task Size
099.go	589.1M	46.5M	12.7	12.7
124.m88ksim	131.2M	13.6M	9.6	9.9
126.gcc	181.9M	15.7M	11.6	22.5
129.compress	38.5M	2.6M	15.0	15.4
130.li	200.4M	28.3M	7.1	14.7
132.ijpeg	594.6M	27.0M	22.0	41.2
134.perl	41.7M	3.9M	10.6	8.6
147.vortex	2958.8M	211.6M	14.0	15.0
			12.8	17.5

Table 3.5: Size of tasks for SPEC CINT95 benchmarks.

SPEC CPU95 Floating Point Benchmark	Dynamic Inst Count	Dynamic Task Count	Average Task Size	Standard Deviation Task Size
101.tomcatv	14036.4M	164.1M	85.5	96.9
102.swim	765.0M	8.7M	87.8	114.6
103.su2cor	1114.0M	10.3M	107.8	179.3
104.hydro2d	1164.8M	29.8M	39.1	112.4
107.mgrid	5905.4M	54.4M	108.6	58.0
110.applu	660.0M	17.3M	38.1	139.9
125.turb3d	20034.8M	484.2M	41.4	41.5
141.apsi	2910.2M	62.2M	46.8	77.1
145.fpppp	446.1M	6.7M	66.5	59.3
146.wave5	3973.8M	70.8M	56.1	69.6
			67.8	94.9

Table 3.6: Size of tasks for SPEC CFP95 benchmarks.

enough and accurate enough to be of use in deciding possible directions for actual processor designs.

3.4.1 Types

There are three main types of simulation that are of use to the processor architect: instruction simulation, cycle simulation, and timing simulation. Each different type of simulation has different speed/accuracy tradeoffs associated with it. The types are given in the order of least detail to most detail, and consequently fastest to slowest. Note that, in general, two orders of magnitude in speed are given up as the level of detail is improved.

3.4.1.1 Instruction-Level

The purpose of instruction simulation is to perform the dynamic operations specified in the static representation of a program. This type of simulation is often extremely fast, on the order of 100,000's to 1,000,000's of instructions simulated per second on a circa 1997 workstation. In particular, it is often used to boot operating systems and debug software before working hardware exists. Such simulators are written in high-level languages, and are easy and inexpensive to develop and maintain. Unfortunately, these types of simulations provide little information in terms of processor microarchitecture design except for program behavior at the instruction level.

3.4.1.2 Cycle-Level

The purpose of cycle simulation is not only to perform the operations specified by a run of a program, but to accurately measure the number of machine cycles performing such operations takes according to the particular design modeled by the simulator. This type of simulation is slow, on the order of 1,000's to 10,000's of instructions simulated per second on a circa 1997 workstation. In particular, it is often used by microarchitects to make decisions about design features and strategies that affect the overall design of a processor. However, due to the level of detail only small programs or parts of programs are usually simulated. In addition, such simulators are usually written in high-level languages (sometimes hardware design languages), but are relatively more difficult and more expensive to develop and maintain.

3.4.1.3 Timing-Level

The purpose of timing simulation is not really to perform the operations specified by a program or to count machine cycles, though it does this nevertheless. Instead, timing simulation is used to obtain detailed estimates of the amount of time required for parts of the design to perform their function. In particular, it is often used to estimate the actual cycle time of the machine and identify critical paths within the design. This type of simulation is extremely slow, on the order of 10's to 100's of instructions simulated per second on a circa 1997 workstation. Only small code fragments are simulated in this environment. Such simulators are written in an extremely detailed fashion using a hardware description language that describes operation at what is called the register transfer level.

3.4.2 Techniques

Usually, the simulation type must be matched to the simulation technique appropriate to support it for a usable simulation environment. Consider the architecture under investigation the *target machine*, and the computer used to perform the evaluation the *host machine*. With these two entities in mind, there are a two widely known methods for evaluating the characteristics described above for the architecture to be studied. These are trace-driven and execution-driven simulations.

3.4.2.1 Trace-Driven

A trace-driven simulation analyzes a trace of program execution gathered via software and/or hardware means from the host while it executes a program. The trace information is used to drive a simulation of a proposed architecture. This technique can be quite effective if the target and the host are similar architectural organizations and if the target does not require information about the interleaving of events that cannot be captured on the trace from the host. In particular, speculation and out-of-order execution are not only difficult to capture on the trace, but are often likely to differ radically between the host and the target. If the instruction set architecture and the overall design of the target is similar to that of the host, a reasonable degree of simulation accuracy is often possible.

3.4.2.2 Execution-Driven

An execution-driven simulation analyzes program execution by actually executing a program is if it was running on the target machine. There are two common methods to perform this type of analysis, as described below.

The first method is to directly execute each instruction of the program on the actual hardware of the host machine with special software and/hardware mechanisms to handle cases where the host and target are different. This technique can be quite accurate if the target and the host are identical or nearly so. To some extent, differences between the target and the host can be modeled by trapping to code that tracks desired events or by inserting such code inline, directly into the target program binary. If this overhead is modest, the execution as measured on the host machine may still be used to model the execution on the target machine. However, when the target and the host machine are radically different and/or the desired events are frequent, both the speed of simulation and the behavior modeled may be undesirable. Another drawback of this approach is that the instruction set architecture of the target must match that of the host.

The second method is to indirectly emulate each instruction of the program with a detailed model of the target machine. In this case, the program runs on the host machine, but within an environment (provided by software, hardware, or a combination of both) that provides the appearance of the target machine. The simulation performs each instruction of a program and models the behavior of the proposed architecture. Such simulation has the advantage of being able to obtain detailed information about the proposed architecture at varying degrees of accuracy since most facets of the target may be emulated. Unfortunately, the increase in accuracy comes at a decrease in speed. In addition, it may be non-trivial to support the operating system, devices, etc that would be required to accurately model the target machine, in terms of both simulation time and complexity. An advantage of this approach is that the instruction set architecture of the target need not match that of the host (although in general the endianness ought to be the same).

3.4.3 Correctness

Using the types and techniques described above, simulators of varying degrees of speed and accuracy may be developed. In developing an experimental framework for a multiscalar processor, both aspects are important, but accuracy is the most critical. Because a multiscalar processor is in many ways a radical departure from existing processor designs, it is necessary to clearly understand where it experiences performance gains and losses, as well as why it exhibits this behavior. However, for the information collected by the simulator to be useful, it must be correct. One of the most difficult aspects of building any simulated environment is ensuring its correctness. The correctness of concern in this experimental framework takes two forms.

First, the instruction and cycle counts as well as any other statistics from simulation ought to be accurate. Here, accuracy is measured with respect to a real implementation of the target architecture. Unfortunately, in this case of a novel processor architecture, no such implementation exists for comparison. Nevertheless, the assumptions made in modeling ought to be good enough as to give a reasonable approximation of what an implementation of the target would produce. For this reason, the simulator instantiates as many components that have a significant effect on the execution of the actual processor as possible and allows their cycle behavior to be configurable. Moreover, modeling most of the components makes it straightforward to model the interaction of the components as well.

Second, not only must cycle and instruction counts as well as other statistics be accurate, the functional behavior of each part of the machine ought to be representative of its actual behavior. This goal presents the problem of how to debug the simulator to figure out when parts of the system are not functioning properly. Often in a processor that does speculation and recovery, improper functioning of the design is masked by the machine's ability to recover from doing wrong work. This resilience makes the job of debugging especially difficult. For instance, an executed program may produce the correct output, but may not have actually executed the instructions in the intended manner. In this case, a functional bug was masked, and likely a performance bug as well.

In general, no solution exists to deal with both of these problems in a rigorous manner. Nevertheless, in an attempt to mitigate both of these problems, the simulator that is the basis of the experimental framework for this work actually consists of two simulators that run side by side. One simulation is an instruction-level simulation; the other is a cycle-level simulation. The instruction-level simulation runs two orders of magnitude faster than the cycle-level simulation, so its added effect is minimal on the overall execution time. In a straightforward integration, two copies of program data are required, one for each simulation. This overhead in terms of memory is somewhat expensive, but not prohibitive.

The idea of running two distinct simulations is to allow them to check one another in as many places as possible. Though multiscalar execution occurs in parallel and out-of-order, it must conform to the serial in-order semantics of the program for each instruction that is executed in the dynamic instruction sequence. While the cycle-level simulator provides the parallel, out-of-order behavior, the instruction-level simulator provides the serial, in-order behavior. Each instruction that executes in the cycle-level simulation is compared in detail against the instruction-level simulation to ensure that the intended execution behavior is provided. Though this approach cannot catch all functional errors and identify all timing irregularities, it can be a useful and powerful technique (even beyond this use, as described below). In practice, it does indeed solve many difficult to detect functional and timing problems.

3.4.4 Performance

Though there is a clear need for detail and correctness, it still must be possible to perform runs of the benchmarks (with the inputs described above) in a reasonable amount of time. A straightforward way to improve the speed of an underlying simulation type and technique chosen is to adapt the well known hardware monitoring technique of sampling to software simulation. This approach has been used successfully for trace-driven cache simulation [45] and processor simulation [74]. The idea behind sampling is to study the execution behavior of selected parts of a program in order to obtain a representative measure of the execution behavior of an entire program. The parts selected may be chosen in a uniform or non-uniform manner as best fits the application being studied.

In the simulation environment described thus far, it is rather straightforward to incorporate sampling. The basic approach may be described as follows. The instruction-level simulator is run for the entire program to maintain the context of the program. The cycle-level simulator is used to sample the cycle behavior of the program at regular intervals, at a frequency that ensures the proper execution characteristics are captured. Some experimentation with this method (in the context of this thesis work) has shown that a uniform 1 to 10 ratio, alternating cycle-level simulation of 100,000 instructions and instruction-level simulation of 1,000,000 instructions, provides nearly an order of magnitude speed improvement with minimal impact on the observed execution behavior.

An important aspect of sampling is whether to provide warm or cold processor state during the periods when execution is not being sampled. Providing warm processor state requires the instruction-level simulator to update critical cycle-level simulator components such as the caches and predictors. Providing cold processor state allows the instruction-level simulator to ignore the cycle-level simulator components. Because any updates involve overhead, the simulation speed improvement using sampling is degraded to some extent. However, this degradation is warranted if it is necessary to attain accurate results. Again, some experimentation (in the context of this thesis work) has shown that having the instruction-level simulator keep all processor components warm while it runs is an important factor in achieving similar, consistent sampled performance with respect to non-sampled runs.

Tables 3.7 and 3.8 compare the performance given as instruction per cycle for simulator configurations with and without sampling for multiscalar processors running a single task per processing unit (one-at-a-time) and running multiple tasks per processing unit all-at-once, respectively, for the SPEC CINT95 benchmarks. Tables 3.9 and 3.10 provide the same information for the SPEC CFP95 benchmarks. In all of the tables, single task per processing unit configurations are given as **m2**, **m4**, **m8**, and **m16** where the number indicates how many processing units; the multiple tasks per processing unit configurations (always for 4 processing units, the m4 part) are given as **m4x2.one** and **m4x2.all** for 2 tasks per processing unit running one-at-a-time and all-at-once as well as **m4x4.one** and **m4x4.all** for 4 tasks per processing unit running one-at-a-time and all-at-once. (The terminology of processing units, single task and multiple tasks per processing unit, as well as one-at-a-time and all-at-once will be discussed in later chapters.)

Overall, the percentage differences between the sampled and non-sampled results are negligible, in the range of 0% to 3% for almost all of the benchmarks. To be specific, this conclusion is based on the percentage differences in harmonic means of performance for different multiscalar configurations. In terms of the harmonic means, the values most prevalently used throughout this work, the performance always differs by less than 2%. In terms of the bias of these percentage differences, the performance is nearly always lower for sampled than for non-sampled. So, the results using with sampling are slightly pessimistic compared to those without sampling. Moreover, the magnitude of these percentage differences can be seen to increase somewhat with the magnitude of processor performance.

3.4.5 Standard Configuration

The standard configuration of a multiscalar processor for simulation used through this thesis is detailed in Table 3.11. Though the simulator may of course be configured to study a variety of multiscalar processor aspects, these parameters are assumed for the various components of the processor unless otherwise noted. In some cases, the terminology used here to describe the configurations is not actually introduced until the corresponding chapter where the aspect of the multiscalar processor being described is studied.

3.4.5.1 Instruction and Data Processing

The standard instruction and data processing configuration uses 2, 4, 8, or 16 processing units running a single task or 4 processing units running 2 or 4 multiple tasks, one-at-a-time or all-at-once. Each processing unit is configured with a 4-wide uniform, out-of-order issue supported by a 16 entry in-flight queue and an 8 entry on-deck queue. The functional unit latencies of the most common operations performed by the processing units (in terms of the instructions executed) are given in Table 3.12. (Again, as above, this terminology will be discussed in later chapters.)

SPEC CPU95 Integer Benchmark	m2 Sample no	m2 Sample yes	m2 Pcnt Diff	m4 Sample no	m4 Sample yes	m4 Pcnt Diff	m8 Sample no	m8 Sample yes	m8 Pcnt Diff	m16 Sample no	m16 Sample yes	m16 Pcnt Diff
099.go	1.31	1.29	-1.5%	1.71	1.69	-1.2%	2.00	1.97	-1.5%	2.11	2.07	-1.9%
124.m88ksim	1.69	1.69	0.0%	2.36	2.35	-0.4%	3.01	2.99	-0.7%	3.77	3.75	-0.5%
126.gcc	1.40	1.41	0.7%	1.92	1.92	0.0%	2.37	2.35	-0.8%	2.65	2.60	-1.9%
129.compress	1.73	1.72	-0.6%	2.44	2.44	0.0%	3.10	3.06	-1.3%	3.58	3.58	0.0%
130.li	1.34	1.33	-0.7%	1.95	1.91	-2.1%	2.69	2.63	-2.2%	3.08	2.98	-3.2%
132.ijpeg	2.39	2.38	-0.4%	3.68	3.67	-0.3%	4.96	4.96	0.0%	5.87	5.85	-0.3%
134.perl	1.59	1.59	0.0%	2.34	2.36	0.9%	3.16	3.19	0.9%	3.88	3.89	0.3%
147.vortex	1.77	1.77	0.0%	2.87	2.87	0.0%	3.99	3.97	-0.5%	4.95	4.90	-1.0%
HMEAN	1.60	1.59	-0.3%	2.29	2.28	-0.5%	2.94	2.92	-0.9%	3.40	3.36	-1.3%

Table 3.7: Comparison of performance (given as instructions per cycle) with and without simulator sampling for the SPEC CINT95 benchmarks. The simulator configurations model 2, 4, 8, and 16 processing units running a single task per processing unit.

SPEC CPU95 Integer Benchmark	m4x2.one Sample no	m4x2.one Sample yes	m4x2.one Pcnt Diff	m4x2.all Sample no	m4x2.all Sample yes	m4x2.all Pcnt Diff	m4x4.one Sample no	m4x4.one Sample yes	m4x4.one Pcnt Diff	m4x4.all Sample no	m4x4.all Sample yes	m4x4.all Pcnt Diff
099.go	1.83	1.80	-1.6%	1.93	1.89	-2.1%	1.83	1.80	-1.6%	1.97	1.93	-2.0%
124.m88ksim	2.70	2.69	-0.4%	2.86	2.84	-0.7%	2.81	2.80	-0.4%	3.34	3.32	-0.6%
126.gcc	2.15	2.13	-0.9%	2.27	2.23	-1.8%	2.19	2.17	-0.9%	2.40	2.34	-2.5%
129.compress	2.79	2.79	0.0%	2.90	2.88	-0.7%	2.84	2.86	0.7%	3.05	3.04	-0.3%
130.li	2.27	2.24	-1.3%	2.53	2.48	-2.0%	2.29	2.25	-1.7%	2.67	2.62	-1.9%
132.ijpeg	4.01	4.01	0.0%	4.28	4.28	0.0%	4.04	4.04	0.0%	4.58	4.57	-0.2%
134.perl	2.72	2.74	0.7%	2.96	2.95	-0.3%	2.78	2.79	0.4%	3.23	3.18	-1.5%
147.vortex	3.40	3.39	-0.3%	3.63	3.64	0.3%	3.57	3.54	-0.8%	4.05	4.10	1.2%
HMEAN	2.59	2.57	-0.6%	2.76	2.73	-1.1%	2.64	2.62	-0.7%	2.96	2.93	-1.2%

Table 3.8: Comparison of performance (given as instructions per cycle) with and without simulator sampling for the SPEC CINT95 benchmarks. The simulator configurations model 4 processing units running multiple tasks per processing unit.

SPEC CPU95 Floating Point Benchmark	Processing Unit Configuration											
	m2			m4			m8			m16		
	Sample		Pcnt Diff	Sample		Pcnt Diff	Sample		Pcnt Diff	Sample		Pcnt Diff
	no	yes		no	yes		no	yes		no	yes	
101.tomcatv	2.14	2.15	0.5%	3.26	3.27	0.3%	4.72	4.78	1.3%	6.67	6.87	3.0%
102.swim	2.11	2.11	0.0%	3.53	3.50	-0.8%	4.65	4.60	-1.1%	5.77	5.67	-1.7%
103.su2cor	2.32	2.32	0.0%	3.62	3.60	-0.6%	4.90	4.91	0.2%	5.28	5.19	-1.7%
104.hydro2d	1.70	1.69	-0.6%	2.71	2.70	-0.4%	4.19	4.17	-0.5%	5.70	5.66	-0.7%
107.mgrid	3.15	3.15	0.0%	5.27	5.27	0.0%	9.30	9.05	-2.7%	15.30	14.79	-3.3%
110.applu	2.07	2.06	-0.5%	3.38	3.36	-0.6%	5.68	5.60	-1.4%	8.38	8.30	-1.0%
125.turb3d	2.50	2.49	-0.4%	4.32	4.31	-0.2%	6.75	6.70	-0.7%	9.27	9.14	-1.4%
141.apsi	1.80	1.80	0.0%	2.48	2.48	0.0%	3.35	3.30	-1.5%	4.15	3.96	-4.6%
145.fpppp	1.85	1.84	-0.5%	2.91	2.87	-1.4%	3.95	3.88	-1.8%	4.39	4.17	-5.0%
146.wave5	2.11	2.11	0.0%	3.38	3.39	0.3%	5.01	5.02	0.2%	6.21	6.28	1.1%
HMEAN	2.11	2.11	-0.2%	3.34	3.33	-0.4%	4.87	4.83	-0.8%	6.21	6.10	-1.8%

Table 3.9: Comparison of performance (given as instructions per cycle) with and without simulator sampling for the SPEC CFP95 benchmarks. The simulator configurations model 2, 4, 8, and 16 processing units running a single task per processing unit.

SPEC CPU95 Floating Point Benchmark	Processing Unit Configuration											
	m4x2.one			m4x2.all			m4x4.one			m4x4.all		
	Sample		Pcnt Diff	Sample		Pcnt Diff	Sample		Pcnt Diff	Sample		Pcnt Diff
	no	yes		no	yes		no	yes		no	yes	
101.tomcatv	3.51	3.56	1.4%	4.38	4.47	2.1%	3.54	3.60	1.7%	5.99	6.19	3.3%
102.swim	3.89	3.85	-1.0%	4.56	4.52	-0.9%	3.96	3.92	-1.0%	5.36	5.30	-1.1%
103.su2cor	3.82	3.76	-1.6%	4.81	4.70	-2.3%	3.81	3.80	-0.3%	5.01	4.93	-1.6%
104.hydro2d	3.10	3.10	0.0%	4.00	3.98	-0.5%	3.14	3.13	-0.3%	4.86	4.83	-0.6%
107.mgrid	5.91	5.82	-1.5%	8.39	8.22	-2.0%	5.93	5.82	-1.9%	10.01	9.89	-1.2%
110.applu	3.68	3.65	-0.8%	5.26	5.20	-1.1%	3.68	3.69	0.3%	6.13	6.25	2.0%
125.turb3d	4.76	4.74	-0.4%	5.92	5.89	-0.5%	4.76	4.74	-0.4%	6.49	6.43	-0.9%
141.apsi	2.63	2.62	-0.4%	3.21	3.17	-1.2%	2.67	2.65	-0.7%	3.68	3.50	-4.9%
145.fpppp	3.25	3.21	-1.2%	3.83	3.76	-1.8%	3.26	3.21	-1.5%	4.02	3.88	-3.5%
146.wave5	3.63	3.63	0.0%	4.77	4.77	0.0%	3.63	3.64	0.3%	5.28	5.27	-0.2%
HMEAN	3.65	3.63	-0.5%	4.62	4.58	-0.8%	3.67	3.66	-0.4%	5.30	5.24	-1.3%

Table 3.10: Comparison of performance (given as instructions per cycle) with and without simulator sampling for the SPEC CFP95 benchmarks. The simulator configurations model 4 processing units running multiple tasks per processing unit.

Processing Units	2, 4, 8, 16 using single task and 4x2, 4x4 using multiple tasks 16 entry in-flight queue, 8 entry on-deck queue 4-wide uniform, out-of-order issue
Inter-Task Predictor	path-based DOLC=7,3,6,8 path register 64K-entry 2-bit counter, 4 target
Intra-Task Predictor	global-pattern-based 16 bit pattern register 64K-entry 2-bit counter, 2 target
Task Cache	1 K-entry, 2-way associative, 64 byte task descriptor, lru 1 cycle hit, 12 cycle miss to unified, 50 cycle miss to main memory shared storage, 1 bank, unified transaction bus no block access for bus lockup no concurrent access
Instruction Cache	64K-byte, 2-way associative, 32 byte block interleaved, lru 1 cycle hit, 10 cycle miss to unified, 50 cycle miss to main memory shared storage / #bank, #bank = #processing unit, crossbar same block access combining for crossbar lockup-free 1 primary miss per bank, 3 secondary per primary
Register File	4 register per cycle 1 cycle latency between adjacent register file
Data Cache	64K-byte, 2-way associative, 32 byte block interleaved, lru 2 cycle hit, 10 cycle miss to unified, 50 cycle miss to main memory shared storage / #bank, #bank = #processing unit, crossbar same block access combining for crossbar lockup-free 8 primary miss per bank, 3 secondary per primary
Address Resolution Buffer	128 entry per bank, 32-way associative, block version byte disambiguation granularity
Memory Dependence Table	16 entry per bank, 16-way associative, lru
Unified Cache	4M-byte, 2-way associative, 128 byte block interleaved, lru 8 cycle + # 4 word transfer * 1 cycle hit, 50 cycle miss to main memory shared storage, 4 bank, split transaction bus no block access combining for bus lockup-free 4 primary miss per bank, 3 secondary per primary
Main Memory	N/A, N/A, 1024 byte interleaved 34 cycle + # 4 word transfer * 2 cycle access shared storage, 4 bank, split transaction bus no block access combining for bus lockup per bank, no concurrent access per bank

Table 3.11: Standard configuration of a multiscalar processor for simulation.

Integer		Floating Point	
Operation	Latency	Operation	Latency
Add/Sub	1	Sgle Prec Add/Sub	2
Shift/Logic	1	Sgle Prec Multiply	4
Multiply	4	Sgle Prec Divide	12
Divide	12	Sgle Prec Compare	2
Mem Load	3	Dble Prec Add/Sub	2
Mem Store	2	Dble Prec Multiply	5
Branch/Jump	1	Dble Prec Divide	18
Set/Compare	1	Dble Prec Comare	2
Move	1	Abs/Neg	1

Table 3.12: Latencies of functional unit operations for simulation.

3.4.5.2 Instruction Supply

The standard instruction supply configuration comprises two aspects, the hierarchical prediction and the instruction memory. The hierarchical predictor consists of a single inter-task predictor (for the whole processor) and multiple intra-task predictors (for each processing unit). The instruction memory consists of a single task cache (for the whole processor) and multiple instruction caches (for all of the processing units).

The inter-task predictor, one for the whole processor, is configured as a path-based design that predicts among 4 targets. It uses a path register that contains D=7 addresses with L=6 bits from the latest address and O=3 bits from other addresses. The supplied address provides C=8 bits from the control flow point being predicted. (This path-based predictor is specified using the DOLC convention because its design is taken from a similar path based predictor proposed by Jacobson, Bennett, Sharma, and Smith [37].) This configuration produces a total of 32 bits which are split into two parts of 16 bits and XORed in order to index the 64K-entry prediction state of 2-bit counters as well as other information used by the predictor.

The intra-task predictor, one for each processing unit, is configured as a global-pattern-based design that predicts among 2 targets. It uses a pattern register that contains N=16 target patterns with B=1 bits from each control flow point for a total of 16 bits. The pattern register is XORed with the low order 16 bits of the supplied address from the control flow point being predicted. This configuration produces 16 bits to index the 64K-entry prediction state of 2-bit counters as well as other information used by the predictor.

The task cache, one for the whole processor, is configured as 1024 entries with 64 byte task descriptor blocks using 2-way associativity and least recently used replacement. The task cache is connected to the processing units via a unified transaction bus. The access latency for the task cache is 1 cycle on hit. A miss to the next level unified cache is an additional 10 cycles without contention; a further miss to the main memory is an additional 50 cycles without contention. The task cache is a lockup cache that allows no concurrent accesses.

The instruction cache, shared among the processing units, is configured as 64K-byte with 32 byte blocks using 2-way associativity and least recently used replacement. As the storage is shared among the processing units, it is divided evenly among a number of banks that is equal to the number of processing units and interleaved by cache block. The processing units are connected to the instruction caches via a crossbar, which allows combining of concurrent accesses to the same cache block.

Each access to the instruction cache is for 4 contiguous instructions from a processing unit for the address given. The access latency for the instruction cache is 1 cycle on hit. A miss to the next level unified cache is an additional 10 cycles without contention; a further miss to the main memory is an additional 50 cycles without contention. The instruction cache is a lockup-free cache that supports unrestricted hits and supports 1 primary misses per bank as well as 3 secondary misses per primary miss

3.4.5.3 Data Supply

The standard data supply configuration comprises two aspects, the register file and the data memory. The register file consists of multiple register files (for each processing unit). The data memory consists of multiple data caches and other specialized components (for all of the processing units).

The individual register files of each processing unit are connected via a unidirectional point-to-point communication ring. Each link of the ring has a bandwidth of 4 registers per cycle with a latency of 1 cycle per transfer. Each register file contains 32 architected integer and floating point registers. The renamed register storage provided is for all intents and purposes unlimited since it supports whatever processing unit configuration is used. The processing unit to register file bandwidth is also assumed to be sufficient to support whatever processing unit configuration is used.

The data cache, shared among the processing units, is configured as 64K-byte with 32 byte blocks using 2-way associativity and least recently used replacement. As the storage is shared among the processing units, it is divided evenly among a number of banks that is equal to the number of processing units and interleaved by cache block. The processing units are connected to the data caches via a crossbar, which allows combining of concurrent accesses to the same cache block.

Each access to the data cache is for a word (or some part of a word) from a processing unit for the address given. The access latency for the data cache is 2 cycles on hit (pipelined where the pipeline is flushed on a miss). A miss to the next level unified cache is an additional 10 cycles without contention; a further miss to the main memory is an additional 50 cycles without contention. The data cache is a lockup-free cache that supports unrestricted hits and supports 8 primary misses per bank as well as 3 secondary misses per primary miss.

Each bank of the data cache is supported by two specialized components, an address resolution buffer and a memory dependence table. Each address resolution buffer is configured as 128 entries with 32 byte cache block versions using 32-way associativity. Each entry contains a number of versions equal to the number of processing units and provides disambiguation among the versions at the granularity of a byte. Each memory dependence table is configured as 16 entries using 16-way associativity. Each entry contains prediction and synchronization information about a dependent load-store pair.

3.4.5.4 Off-Processor Memory

The standard configuration of the off-processor memory comprises two aspects, the unified cache (that directly supports the task cache, the instruction cache, and the data cache) and the main memory. The off-processor memory is not studied explicitly in this thesis, though it is assumed implicitly as an underpinning that supports the other processor aspects that are studied.

The unified cache (shared among the task cache, instruction cache, and data cache) is configured as 4M-byte with 128 byte blocks using 2-way associativity and least recently used replacement. As the storage is shared, it is divided evenly among 4 banks and interleaved by cache block. The task cache, instruction cache, and data cache are connected to the unified cache via a split transaction bus.

The main memory is connected to the unified cache via a split transaction bus. The main memory is divided into 4 banks interleaved by the page size (4 K-byte). Though the banks of the main memory may have concurrent accesses among them, there are no concurrent accesses per bank. Each access (miss or writeback) to the main memory from the unified cache is for a cache block for the address given. The access latency for the main memory is 50 cycles without contention.

3.5 Summary

This chapter presented the experimental framework that is used for the study of a multiscalar processor performed in this thesis.

The methodology used to study multiscalar processors provides an execution environment that is a combination of software and hardware. The principle components are a production-quality compiler that produces highly optimized code (the software) and a simulator that models a wide range of processor designs as well as provides and gathers many statistics on behavior and performance (the hardware).

The compiler is a modified version of GNU GCC 2.7.2 targeted to an enhanced version of the MIPS instruction set architecture that does not include delay slots of any kind. The simulator is execution-driven and models a multiscalar processor at the instruction-level as well as the cycle-level. A detailed description of the standard configuration used throughout this thesis was given above.

In addition to the tools used to study multiscalar processors, this chapter also presented the benchmark programs, the SPEC CPU95 (or just SPEC95) suite. This benchmark suite comprises both integer and floating point programs which represent a range of different types of applications. Moreover, it is an apt choice for the purpose of this work since it is recognized as a standard industry benchmark suite that provides a measure of compute-intensive performance for the processor, memory hierarchy, and compiler.

Chapter 4

Instruction and Data Processing

This chapter studies the instruction and data processing of a multiscalar processor. Section 4.1 gives an overview of how tasks and processing units are used to perform instruction and data processing. Section 4.2 explains how instruction-level parallelism is extracted in a multiscalar processor using tasks and processing units. Section 4.3 identifies the processing phases with respect to tasks that support the multiscalar model. Section 4.4 discusses the interaction of the processing phases with the organization of tasks and processing units. Section 4.5 investigates the design of the processing units that perform instruction and data processing. Section 4.6 provides an evaluation of these designs. Section 4.7 gives a summary of this chapter.

4.1 Overview

The instruction and data processing in a multiscalar processor is performed by a collection of processing units. A processing unit is comparable to the execution core of a conventional processor. It provides the execution resources (pipeline, functional units, reservations stations, reorder buffer, *etc.*) to perform the computation of a task. The multiple processing units of a multiscalar processor, though separate, act as a single, aggregate processor. This appearance is achieved by handling instruction and data processing for tasks so as to maintain the appearance of program order among them.

4.1.1 Basics

A processing unit executes instructions at a task granularity. A task is assigned to a processing unit for the lifetime of its execution. It executes all instructions on the assigned processing unit using only the resources available to that processing unit. The execution of tasks is speculative with respect to both the control and data relationships between instructions. Thus, each processing unit buffers the modifications made during task execution to speculative state and updates the architectural state when the control and data relationships have been resolved.

To mimic the program order that must be maintained among tasks, the tasks and processing units may be organized as queue-like structures (either explicitly or implicitly), as shown in Figure 4.1. The tasks are organized in an explicit task queue. That is, tasks are incorporated into a queue structure as predicted – with head and tail pointers indicating the oldest and newest tasks – in order to maintain the program order among them. The tasks are assigned to processing units one-to-one and mapped in numerical round-robin order, wrapping around when the highest numbered processing unit has been assigned a task. In addition, tasks are only removed from the queue at the head and added to the queue at the tail.

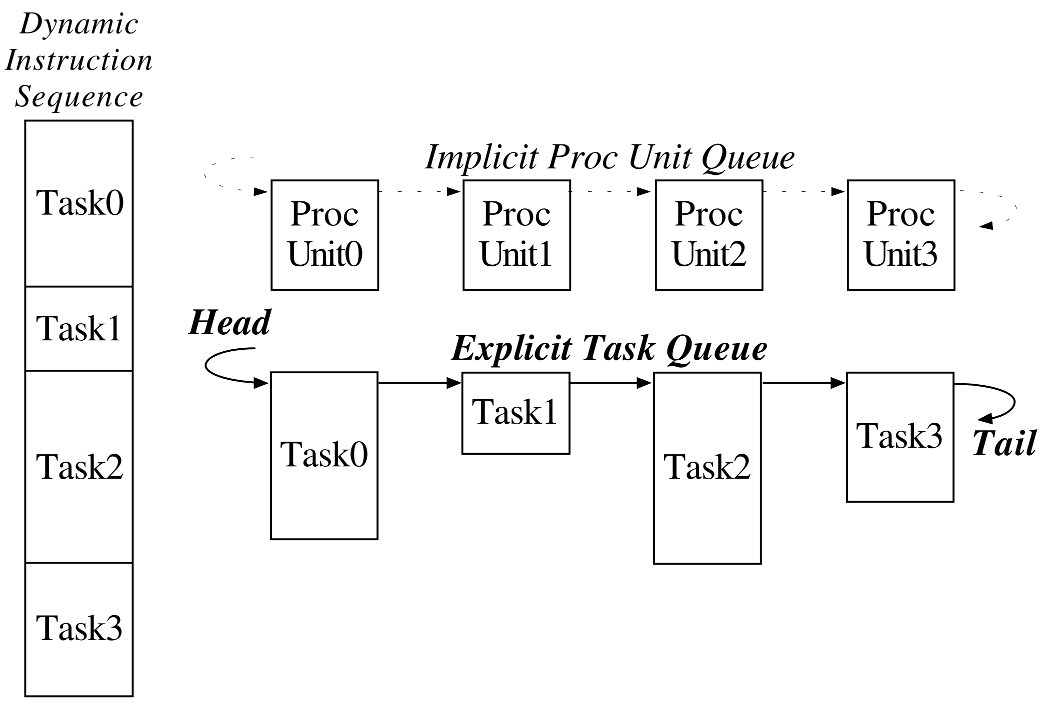


Figure 4.1: The organization of tasks and processing units in a multiscalar processor.

Using this approach for the tasks, an implicit processing unit queue naturally arises because the processing units are subsequently allocated and deallocated in the same queue-like manner as tasks are added and removed from the explicit task queue. An advantage of providing an implicit processing unit order that matches the explicit task order is that there is no need to resort to tag comparisons to match the order among tasks and processing units. Though this organization is straightforward, its lack of flexibility means that a multiscalar processor may suffer from performance losses due to load balance, limited exposed parallelism, and/or resource under-utilization.

4.1.2 Insights

The study of instruction and data processing provided in this chapter explores the interaction between the tasks and the processing units that execute them in order to investigate these sources of performance loss. In general, the amount of instruction-level parallelism that a multiscalar processor is able to expose and exploit is influenced by the number and/or size of the tasks that make up its instruction window as well as the manner in which the processing units execute the instructions within the instruction window. In particular, the performance that a multiscalar processor is capable of attaining with respect to its instruction window is determined by various inter-task and intra-task aspects of the processing unit design.

4.1.2.1 Inter-Task Aspect

For the inter-task aspect, this study addresses the mapping policy, which dictates how tasks are assigned to processing units, and the running policy, which dictates how many tasks a processing unit may run concurrently, as the key design choices. In terms of mapping policy, two policies – round-robin and back-to-back – are described, but only the former is evaluated. In terms of the running policy, two policies – one-at-a-time and all-at-once – are described and evaluated. A variety of different organizations that support up to a total of 16 tasks and 16 processing units, running either a single task or multiple tasks per processing unit, are considered to offer different cost-performance points.

Considering organizations that run a single task per processing unit, linear performance improvements may be achieved as the number of tasks in concurrent execution are doubled, assuming that tasks map round-robin and run one-at-a-time on the processing units. For SPEC CINT95 programs, the harmonic mean of the instructions per cycle ranges roughly from 1.50 to 3.50, using from 2 to 16 tasks and, as a result of the one-to-one assignment, an equal number of processing units. For SPEC CFP95 programs, the harmonic mean of the instructions per cycle ranges roughly from 2.00 to 6.00, using from 2 to 16 tasks and likewise an equal number of processing units.

Considering organizations that run multiple tasks per processing unit, many different designs using more tasks with less processing units or less tasks with more processing units are possible. Yet, a configuration of 4 processing units running 4 tasks all-at-once per processing unit (a total of 16 tasks) performs best and is able to nearly match (90%) the performance of the configuration with the highest performance, 16 processing units running 1 task per processing unit, but at a fraction of the cost (25%). In particular, for the SPEC CINT95 programs, the harmonic mean of the instructions per cycle reached roughly 3.00 (as compared to 3.50); for the SPEC CFP95 programs, the harmonic mean of the instructions per cycle reached roughly 5.50 (as compared to 6.00).

4.1.2.2 Intra-Task Aspect

For the intra-task aspect, this study addresses the instruction window, the instruction selection and issue, as well as the instruction execution characteristics of the individual processing units as key design choices. The performance improvements that may be realized using a multiscalar processor depend to a significant extent on these particular microarchitectural characteristics of the processing units. With respect to the results of this study, the overall significance of these factors in terms of their impact on performance may be ranked roughly as follows: the instruction selection and issue (first), the instruction execution (second), and the instruction window (third). Clearly, all of these factors are inter-related in a processing unit design choice. However, the relative performance ranges for the various configurations of each factor support this relative ordering. The following description focuses on the individual characteristics of each of the factors.

Considering the instruction selection and issue characteristics of the processing units, performance grows steadily, for the SPEC CINT95 and SPEC CFP95 programs among both in-order and out-of-order designs, with increases in issue width for all multiscalar processor configurations (1, 2, 4, 8, and 16 processing units) studied. Nonetheless, the rate of growth rate going from 1-wide to 2-wide is more significant, roughly a 30% improvement, than going from 2-wide to 4-wide, roughly 10% improvement, for both integer and floating point benchmarks. Moreover, in absolute terms, the out-of-order designs have approximately a 20% performance advantage over in-order designs of equal issue width. Due to the combined effects of absolute performance difference and performance growth rate, there is a crossover point (at 2-wide out-of-order and 4-wide in-order) where a lower issue width out-of-order processing unit performs better than a higher issue width in-order processing unit; the difference at this crossover point is around 15%.

Considering the instruction execution characteristics of the processing units, the number and type (classified as int/fp compute, memory, and control) of functional units associated with each processing unit can have a direct impact on performance for multiple instruction per cycle issue widths on both SPEC CINT95 and SPEC CFP95 programs. For issue widths greater than 1 instruction per cycle (where number and type is actually a factor), a functional unit configuration of 2 compute, 2 memory, and 1 control is needed to avoid constraining execution and overall performance. Furthermore, using more than 2 of any type of functional unit appears to offer no appreciable performance advantage. Moreover, the results of this study indicate that one control functional unit is always sufficient, and that the number of memory functional units is a somewhat more important factor than the number of compute functions – with a key result being a 10% performance loss for 2-wide (out-of-order) issue and an 15% performance loss for 4-wide (out-of-order) issue when the number of memory functional units is decreased from 2 to 1.

Considering the instruction window characteristics of the processing units, the size of the instruction window for each processing unit in a multiscalar processor has only small impact on overall performance (unlike a conventional superscalar processors where it has a large impact). With respect to the results of this study, a multiscalar processor with processing units that support an instruction window of only 16 instructions in-flight (decoded and issued) and 8 instructions on-deck (decoded but not issued) achieves comparable performance to one with processing units that support an instruction window of 64 instructions in-flight and 64 instructions on-deck. In particular, for SPEC CINT95 programs, the performance is identical, and for the SPEC CFP95 programs, the performance is only about 10% different comparing these small and large window designs. A key point of this study is that small window structures, not only may support higher clock speeds than large window structures, but may also attain levels of instruction-level parallelism comparable to large window structures for multiscalar processors, despite this difference in size.

4.2 Instruction-Level Parallelism

With the overview of the instruction and data processing just presented in mind, it is worth switching gears to consider how the characteristics of the task and processing unit organizations into performance in terms of instruction-level parallelism.

Recalling its basic approach to instruction and data processing, a multiscalar processor extracts instruction-level parallelism from a serial program by dividing it into tasks and executing these tasks in parallel on the different processing units of the processor. The tasks provide the window on the dynamic instruction sequence from which a multiscalar processor may extract instruction-level parallelism to improve performance. The processing units provide the execution engine on which this performance is obtained. A key aspect of performing instruction and data processing is the interdependence between the characteristics of the tasks and the characteristics of the processing units with regard to extracting instruction-level parallelism.

For a multiscalar processor, the aggregate performance in terms of instruction-level parallelism obtained from all tasks is the sum of the separate performances obtained by each processing unit from its task. The usual means by which performance is improved in terms of instruction-level parallelism is performing instruction and data processing with wider issue in a bigger window. The wider issue is used to be able to perform more parallel operations at the same time. The bigger window is used to be able to find more such operations to perform. There are four interrelated factors involved in determining the issue and window capabilities of a multiscalar processor: the number of tasks and the size of the tasks, and the number of processing units and the power of the processing units.

The number of tasks and the size of each task dictate the size of the exposed window on the dynamic instruction sequence that a multiscalar processor may provide. By increasing the number of tasks, the size of each task, or both, the size of the exposed window may be increased. The extent to which the exposed window may be exploited is determined by the number of processing units and the power of each processing unit – the amount of instructions that may be issued each cycle, whether the instructions may be selected out of program order, and how far into the task to look ahead for the instructions. By increasing the number of processing units, the power of each processing unit, or both, the extent to which the exposed window is exploited may be increased.

4.3 Processing Phases

In order to understand where performance gains and losses occur in terms of instruction-level parallelism, it is useful to break down instruction and data processing of tasks into its constituent phases. The processing of a task may be described as consisting of several phases (similar to the handling of an instruction in a pipeline) given as follows: predict, assign, execute, resolve, commit, and perhaps squash. These phases account for the period of instruction and data processing that takes place in the multiscalar processor from the entry to the exit of a task. Within each task, these phases occur in the order given, except for squash which may occur at any time during any phase (due to a dependence violation). Among tasks, the predict and assign phases are performed in program order so tasks may be entered in program order, and the commit phases are performed in program order so tasks may be exited in program order. The advantage of performing these phases in program order is that maintaining serial semantics, which otherwise might be quite complex, is simple. The other processing phases need not be ordered among tasks (only within tasks).

Figure 4.2 shows the processing phases as tasks run in parallel on the processing units.

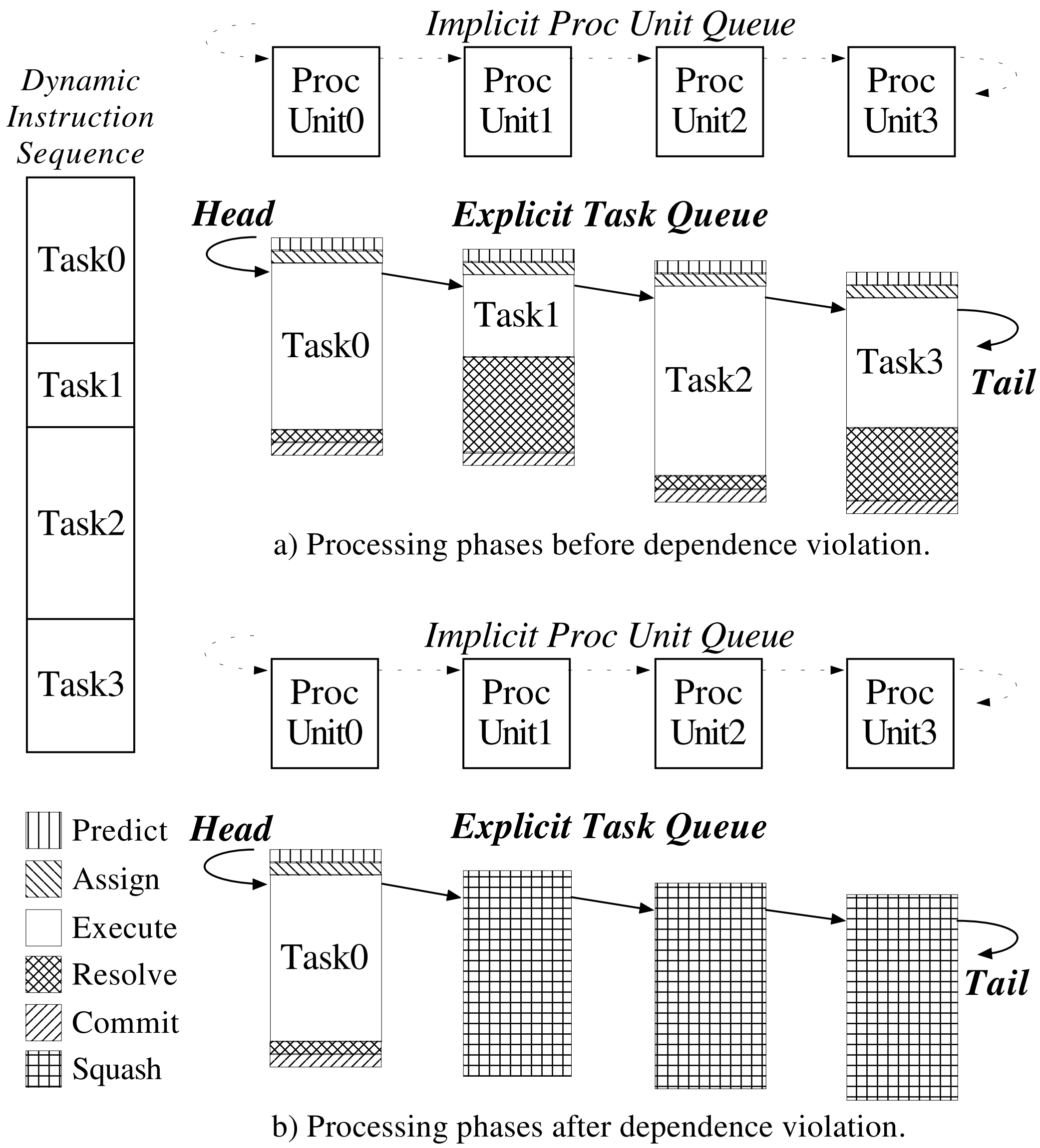


Figure 4.2: The task processing phases that support the multiscalar execution model.

Each phase is identified by its own shading pattern. For the purpose of illustration, it is assumed that 4 tasks run in parallel on 4 processing units, and that the processing of all of the tasks occurs at the same time and until all of the tasks have performed all phases of processing. An actual multiscalar processor would not only begin new tasks in a pipelined fashion, it would begin a new task at the end of the processing for an old task (not shown in the figure). It is worth noting that instruction-level parallelism is exploited only during the execute phase. During all of the other phases no instruction-level parallelism is exploited because each of these phases represents overhead in some form or another associated with handling computation in the form of tasks.

4.3.1 Predict

The predict phase is the part of processing in which the next task is predicted and is incorporated into the explicit task queue. At this point this next task has not actually been associated with a particular processing unit. (This association is made in the next phase, when the task is assigned to a processing unit.) As shown in part a) of Figure 4.2, it is a relatively small part of overall task processing. The actions associated with this processing phase must occur in-order among tasks. That is, the next task cannot perform them until after the previous task has performed them.

4.3.2 Assign

The assign phase is the part of processing in which the oldest task in the explicit task queue that has not been associated with a processing unit is assigned to a processing unit for execution. (If only one task is predicted at a time, then the oldest task not yet associated with a processing unit is the newest task predicted.) As shown in part a) of Figure 4.2, it is a relatively small part of overall task processing. The actions associated with this processing phase must occur in-order among tasks. That is, the next task cannot perform them until after the previous task has performed them.

4.3.3 Execute

The execute phase is the part of processing in which the instructions that constitute the computation of a task are executed. Though the execute phase is the only phase where processing extracts instruction-level parallelism, even in this phase there may be stalls due to control and/or dependences within and among tasks. As shown in part a) of Figure 4.2, it is a relatively large part of overall task processing (as it must be to keep overhead losses from dominating instruction-level parallelism gains) so long as the task size is relatively large. The actions associated with this processing phase may (and usually do) occur out-of-order among task. Thus, a task can perform them before, during, as well as after the next and/or previous tasks perform them.

4.3.4 Resolve

The resolve phase is the part of processing in which the control and data dependences that may have been speculated during the execution of a task are resolved. At this point, all of the instructions of a task have been executed. The update from the speculative to the architectural state is all that remains, but it cannot be performed until it is certain that all control and data dependences have been resolved. As shown in part a) of Figure 4.2, it may be a relatively small part, as for **Task0** and **Task2**, or a relatively large part, as for **Task1** and **Task3**, of overall task processing, depending on the situation. The actions associated with this processing phase may (and usually do) occur out-of-order among task. Thus, a task can perform them before, during, as well as after the next and/or previous tasks perform them.

4.3.5 Commit

The commit phase is the part of processing in which the update from the speculative to the architectural state is performed. This update may occur once all earlier control and data dependences have been resolved. In general, these dependences are known to be resolved after all earlier tasks have performed their commit phases, and the task has become the head of the explicit task queue. In part a) of Figure 4.2, it is shown as being a relatively small part of overall task processing. However, it might be a relatively large part, depending on how the updates are performed. The actions associated with this processing phase must occur in-order among tasks. That is, the next task cannot perform them until after the previous task has performed them.

4.3.6 Squash

The squash phase is the part of processing in which any incorrect speculative execution that may have occurred is discarded, and the processor is reset to the proper state to continue execution. All of the other phases of processing are performed in the order described. However, the squash phase may occur at any time during the other phases in response to a dependence violation. In part b) of Figure 4.2, a dependence violation at the end of **Task0** has converted all phases in **Task1**, **Task2**, and **Task3** beyond the dependence violation into the squash phase. Though the squash phase is an insignificant part of overall task processing in the absence of dependence violations, it may be a significant part in their presence, whether the dependence violations are due to incorrect control as well as data speculation.

4.4 Concerns/Challenges

To improve performance, a multiscalar processor needs to maximize the amount of time it spends extracting instruction-level parallelism and to minimize the amount of time it spends handling task overhead. In terms of the processing phases, a multiscalar processor needs the execute phase to dominate the other phases in terms of overall time. This condition may not guarantee a multiscalar processor extracts instruction-level parallelism, but it does guarantee that it has the opportunity to do so. In general, the processing phases do exhibit behavior where task overhead takes away from useful execution.

For the discussion that follows, the specific example of Figure 4.2 is used to make this point. In part a), the predict and assign phases of processing as well as the commit phase of processing are relatively small. The squash phase shown in part b) of the figure is relatively big, but is usually avoided with proper handling of control and data dependences. Under the assumption that dependence violations are infrequent, the squash phase is not of much concern here. However, the fact that the resolve phase is often nearly as big (and sometimes bigger) than the execute phase, as for **Task1** and **Task3** in the figure, is of particular concern.

The relationship between the resolve and execute phases may be attributed to the combination of assigning tasks to processing units one-to-one and performing the commit phase of processing in program order. There is an interaction between these aspects of a multiscalar processor that may represent a significant obstacle to exposing and exploiting instruction-level parallelism if not handled properly. Of course, there are means of overcoming this obstacle, but it is necessary to understand the concern before factoring these means into the design of a multiscalar processor.

4.4.1 Exposing Instruction-Level Parallelism

Once the number of processing units is fixed and tasks are assigned one-to-one to them, the exposed window on the dynamic instruction sequence is determined by the size of the tasks. If the tasks are big, as in the floating point benchmarks described in Chapter 3, then few processing units may expose a big window. However, if the tasks are small, as in the integer benchmarks described in Chapter 3, then many processing units may be needed to expose a big window.

With an average task size of approximately 60 instructions for the floating point benchmarks, 4 processing units may expose a window of approximately 240 instructions (significantly bigger than the window exposed by conventional processors). With an average task size of approximately 10 instructions for the integer benchmarks, 4 processing units may expose a window of approximately 40 instructions (comparable to the window exposed by conventional processors).

There are primarily two ways to increase the size of the exposed window for instruction-level parallelism: run the same number of bigger tasks or run more tasks (as many as needed according to the task size).

It may not be straightforward to simply divide a program into bigger tasks, since there are both software and hardware aspects involved with respect to this process. For instance, bigger tasks may require more task exits than can be specified by the compiler or handled by the processor. In addition, a more fundamental difficulty is that the size of the tasks is influenced by the program from which the tasks are made. Some programs may be well suited for providing big tasks (such as floating point codes), while others may be well suited for providing small tasks (such as integer codes). Thus, it may be that the actual size of tasks cannot be manipulated in an arbitrary manner.

Assuming that manipulating task sizes is not a general means to increase the size of the exposed window, the other straightforward alternative of simply running more tasks may be appropriate. However, running more tasks is not without complications. Due to the assigning of tasks to processing units one-to-one, it is necessary to provide more processing units to run more tasks. In particular, a processing unit must be provided for each task. In the worst case, this approach may require the use of perhaps a huge number of the processing units. For example, while only ten tasks would be needed to open a thousand instruction window if each task was a hundred instructions, a hundred tasks would be needed if the tasks were ten instructions.

4.4.2 Exploiting Instruction-Level Parallelism

It might be that providing more processing units to run more tasks solves any problems with respect to exposing instruction-level parallelism. Yet, even if the cost of providing additional processing units is acceptable – it may be an order of magnitude in the situation described above – the assigning of tasks to processing units one-to-one in conjunction with performing the commit phase of processing in program order may still pose difficulty with respect to exploiting instruction-level parallelism.

If there is variability in task sizes and/or execution times, small tasks may go to their resolve phase well in advance of big tasks. However, because the commit phases must be performed in program order among tasks, a task remains in its resolve phase until all earlier tasks have gone through their commit phases. As is the case for **Task1** and **Task3** in part a) of Figure 4.2, the resolve phases are comparable in size to the execute phases for the tasks. Yet, during its resolve phase no instruction-level parallelism may be exploited from a task. If the other tasks in their execute phases were available, the non-useful resolve phases might be overlapped with other useful execute phases.

Even if tasks remain in their execute phases, the extent to which instruction-level parallelism may be exploited might be limited by dependences within or among tasks that stall execution on processing units. In this case, there may be resources available that might be used to exploit instruction-level parallelism from other tasks, but that cannot be used in this way due to the one-to-one assigning of tasks to processing units. Yet, even if the execution is not stalled on the processing unit, the capacity for instruction-level parallelism on processing units might not be fully utilized by one task (similar to what has been observed in conventional processors [94]).

4.5 Processing Unit Design

In terms of processing unit design there are obvious considerations with respect to how many processing units to provide and what type of processing units to provide. However, with respect to the concerns/challenges, the two specific difficulties in exposing and exploiting instruction-level parallelism need to be addressed as part of processing unit design. The first is the difficulty associated with performing the commit phase of processing in program order among tasks. The second is the difficulty associated with assigning tasks to processing units one-to-one.

This section addresses the concerns/challenges for each of these problems in order to consider how to pursue a processing unit design that is free from obvious performance limitations. Two solutions are proposed to solve the problems. Then, the most promising solution is described in relation to how it fits into the processing unit design. This description presents the solution in two parts, according to whether it factors into extracting instruction-level parallelism from an inter-task perspective (among tasks) or from an intra-task perspective (within tasks).

4.5.1 Concerns/Challenges

The way that processing unit design might handle these two difficulties is to solve either of the problems or to solve both of the problems. In order to choose among these options, it is necessary to consider what design changes are involved as well as what problems are solved by such design changes.

Though it may be possible to consider allowing tasks to perform their commit phases out of program order to avoid unproductive waiting in their resolve phases, a mechanism to do so merely provides a level of indirection. In some manner, the speculative to architectural state updates must be performed in program order to guarantee precise interrupts. Moreover, the organization (described earlier) of tasks and processing units that maintains order among them is sure to be disrupted by this approach. As old tasks may commit at any point in the explicit task queue, new tasks must be assigned to arbitrary processing units. As a result, the ordering that provided the implicit processing unit queue is broken, so an alternative means of ordering tasks and processing units is required. Such a change is likely to make maintaining serial semantics more complex. Moreover, this solution does not address the window size difficulty of running more tasks without providing equally more processing units, nor does it address the difficulty that processing units may be unable to exploit enough instruction-level parallelism in their execute phases to be fully utilized.

Rather than change the fundamental aspect of handling the commit phase of tasks, it may be more straightforward to consider breaking the one-to-one assignment by executing multiple tasks rather than a single task on each processing unit – providing a many-to-one assignment. This approach addresses the window size difficulty directly by allowing as many tasks as can be supported on each processing unit to run. It addresses the difficulty of under-utilized processing units by providing more tasks and hence more instructions from which to exploit instruction-level parallelism. This approach addresses the unproductive waiting by tasks in their resolve phases indirectly by allowing tasks to overlap such idle time with useful work, rather than by performing the commit phases of processing out of program order. Though changing the assignment of tasks to processing units offers a more complete solution to these problems than does performing commit phases out of program order, it is not a panacea. Ultimately, the number of tasks that may be assigned to a processing must be fixed, meaning the extent to which the many-to-one assignment may combat difficulties exposing and exploiting instruction-level parallelism must be limited. The use of many-to-one assignment may be viewed as a special case (for multiscalar processors) of the general concept of load balance (for multiprocessors [52]).

4.5.2 Inter-Task Perspective

From the inter-task perspective, this study of processing unit design focuses on how to extract as much instruction-level parallelism as possible among tasks using many-to-one assignment of tasks to processing units.

4.5.2.1 Many-To-One Mechanism

The concept of many-to-one assignment of tasks to processing units may be handled via the incorporation of a multithreading mechanism into the design of the processing units [43, 94]. Using multithreading, the cost of supporting many tasks is proportional to the cost of providing additional resources for each task within a processing unit, rather than to the cost of providing an additional processing unit for each task. So long as the overheads associated with multithreading can be contained, and the cost of the additional resources is small relative to the cost of an additional processing unit, this many-to-one mechanism is an attractive one from a cost-performance perspective.

The effective use of multithreading in a multiscalar processor, though, still requires that the fundamental difference between multiscalar processors and conventional multithreaded processors be addressed. That is, unlike most uses of multithreading, the threads (*vis-a-vis* tasks) of a multiscalar processor are not independent. However, the multiscalar mechanisms can be relied upon to handle the dependences between tasks, so long as the application of multithreading to the processing units remains within the framework of the multiscalar model.

The policy with regard to running tasks on processing units for the multithreading mechanism is not of particular concern for the multiscalar model, so long as it only involves the amount of concurrency that is supported between the tasks assigned to a processing units. The multithreading mechanism must ensure that tasks do not interfere with one another and are executed as if independent, since the multiscalar model can ensure that control and data dependences among tasks are handled properly if this behavior is provided.

The policy with regard to mapping tasks to processing units for the multithreading mechanism is of concern due to assumptions in the multiscalar model that not only ensure the dependences among tasks are handled properly, but also ensure that the dependences may be handled in a straightforward manner. Therefore, it is advantageous to restrict the multithreading policy of which tasks are mapped to which processing units, such that the organization of tasks and processing units (already described) is not disrupted. Otherwise, handling control and data dependences might become complicated.

4.5.2.2 Mapping Policy

Though the assignment of tasks to processing units may be many-to-one, the policy about which tasks are mapped to which processing units requires further consideration.

The mapping policy does not determine that tasks are assigned to processing units many-to-one, but instead it determines which tasks may be mapped to which processing units in the many-to-one assignment. For a one-to-one assignment of tasks and processing units, the situation is straightforward. The tasks are assigned to processing units one-to-one and mapped in round-robin numerical order, wrapping around when the highest numbered processing unit has been assigned a task (as described earlier). Recall that the advantage of this policy is that the order among tasks and processing units may be maintained easily.

For the many-to-one mapping of tasks and processing units, the situation is more involved. Though the mapping might be performed arbitrarily, this policy is undesirable because it breaks down the organization of explicit task ordering and implicit processing unit ordering that allows control and data dependences to be handled in a straightforward manner. The key constraint that preserves such an organization is that adjacent tasks must be mapped to adjacent processing units or to the same processing unit. Even with this constraint, a range of different mapping policies are possible. However, only the two simple policies, round-robin and back-to-back, shown in Figure 4.3, are considered here.

The round-robin mapping policy is an extension of the mapping policy used for one-to-one assignment. It performs the same mapping of tasks to processing units in numerical round-robin order, wrapping around as needed. However, in contrast to one-to-one assignment, this process does not stop once each processing unit has been assigned one task. Instead, this process continues until each processing unit has been assigned as many tasks as it can handle. As illustrated in part a) of Figure 4.3, not only is **Task0** mapped to **Proc Unit0**, so is **Task4**. Similarly, **Task1** and **Task5** are mapped to **Proc Unit1**, **Task2** and **Task6** are mapped to **Proc Unit2**, and **Task3** and **Task7** are mapped to **Proc Unit3**. For this mapping policy, concurrent execution with the next processing unit may begin as soon as the current processing unit has been assigned one of its tasks.

The back-to-back mapping policy is also an extension of the mapping policy used for one-to-one assignment. It performs the mapping of tasks to processing units in numerical order, wrapping around as needed. However, rather than assign only one task as the process moves from processing unit to processing unit, it assigns as many tasks as a processing unit can handle. Thereby, a consecutive sequence of tasks is mapped to each processing unit. As illustrated in part b) of Figure 4.3, not only is **Task0** mapped to **Proc Unit0**, so is **Task1**. Similarly, **Task2** and **Task3** are mapped to **Proc Unit1**, **Task4** and **Task5** are mapped to **Proc Unit2**, and **Task6** and **Task7** are mapped to **Proc Unit3**. For this mapping policy, concurrent execution with the next processing unit may not begin until the current processing unit has been assigned all of its tasks.

The round-robin mapping policy spreads the dynamic instruction sequence across the processing units. In contrast, the back-to-back mapping policy clusters the dynamic instruction sequence on processing units. Moreover, unlike the back-to-back mapping policy, the round-robin mapping policy results in the same tasks running on the same processing units for many-to-one assignment as for one-to-one assignment. If the compiler is successful at dividing the dynamic instruction sequence into tasks that are mostly independent, then the back-to-back policy might tend to serialize execution (working against the compiler), while the round-robin policy might tend to parallelize execution (working with the compiler). In light of these observations and the fact that initiation of concurrent execution may be delayed for the back-to-back policy as compared to the round-robin policy, only the round-robin policy is used for this study.

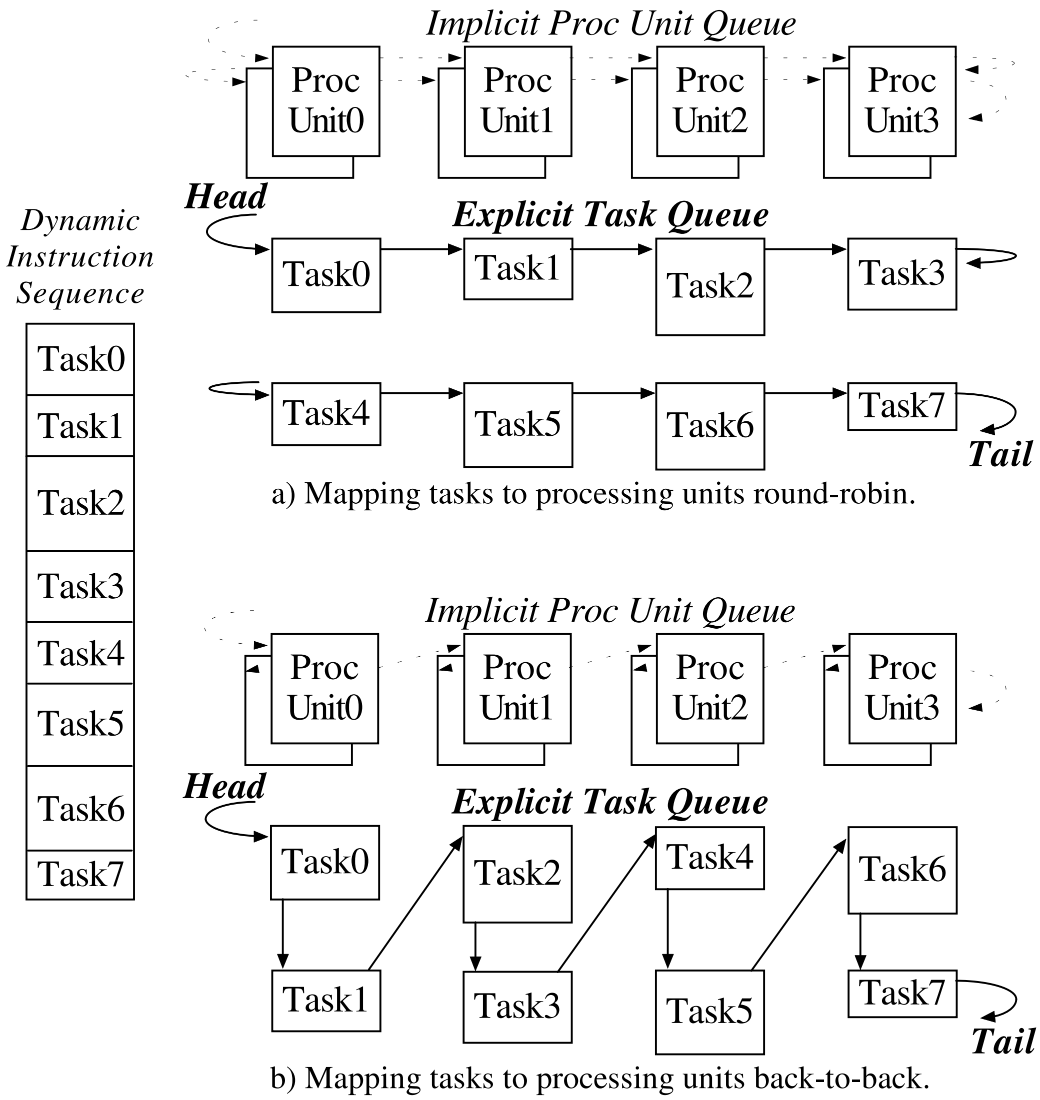


Figure 4.3: Mapping multiple tasks to each processing unit.

4.5.2.3 Running Policy

The running policy determines the amount of concurrency with respect to the overlap of the tasks. Lower concurrency might mean lower cost with lower performance. Higher concurrency might mean higher cost with higher performance. Because an actual design probably lies somewhere in between low cost and high cost, it is useful to consider what might be achieved at the extremes in order to set a range for this policy choice. The two ends of the running policy spectrum are shown in Figure 4.4. The conservative, low cost approach shown in part a) of the figure is called one-at-a-time. The aggressive, high cost approach shown in part b) of the figure is called all-at-once. Both approaches use the round-robin mapping policy (described above). Though this study limits its consideration to the following two approaches, many others (*e.g.* switching tasks on instruction or data cache misses and even control or data mispredictions) are possible, making this area of multiscalar processing a fertile one for further consideration.

The one-at-a-time running policy does not allow any overlap of task execute phases. However, other processing phases may be overlapped, except for the predict and assign phases as well as the commit phases which still must be performed in program order. It primarily attacks the problem of unproductive resolve phase waiting (load balance), and secondarily attacks the problem of limited exposed window size (parallelism). It has no effect on the under-utilization of processing units during the execute phase (since tasks do not execute at the same time). As can be seen in part a) of Figure 4.4, the one-at-a-time policy allows the execute phases of later task to overlap the resolve phases of earlier tasks. For this policy, older tasks are always run before newer tasks to ensure forward progress. This policy is considered conservative because it seems to imply relatively straightforward changes be made to a conventional processing core. It may be described as essentially a form of coarse-grain multithreading at the task level.

The all-at-once running policy allows overlap of all task processing phases, except for the predict and assign phases as well as the commit phases which still must be performed in program order. It handles all problems considered as concerns/challenges. To be specific, it attacks the problems of unproductive resolve phase waiting (load balance), limited exposed window size (parallelism), as well as under-utilization of processing units during the execute phase (since tasks execute at the same time). As can be seen in part b) of Figure 4.4, the all-at-once policy allows the execute phases of earlier and later tasks to overlap. Likewise, it allows the execute phases of later to tasks to overlap the resolve phases of earlier tasks. For this policy, older tasks always have priority over newer tasks for any processing unit resources to ensure forward progress and to avoid undue delay of older tasks by newer tasks. This policy is considered aggressive because it seems to imply relatively more involved changes be made to a conventional processing core. It may be described as essentially a form of processor coupling [43] or simultaneous multithreading [94].

4.5.3 Intra-Task Perspective

From the intra-task perspective, this study of processing unit design focuses on how to extract as much instruction-level parallelism as possible within each task running on a processing unit

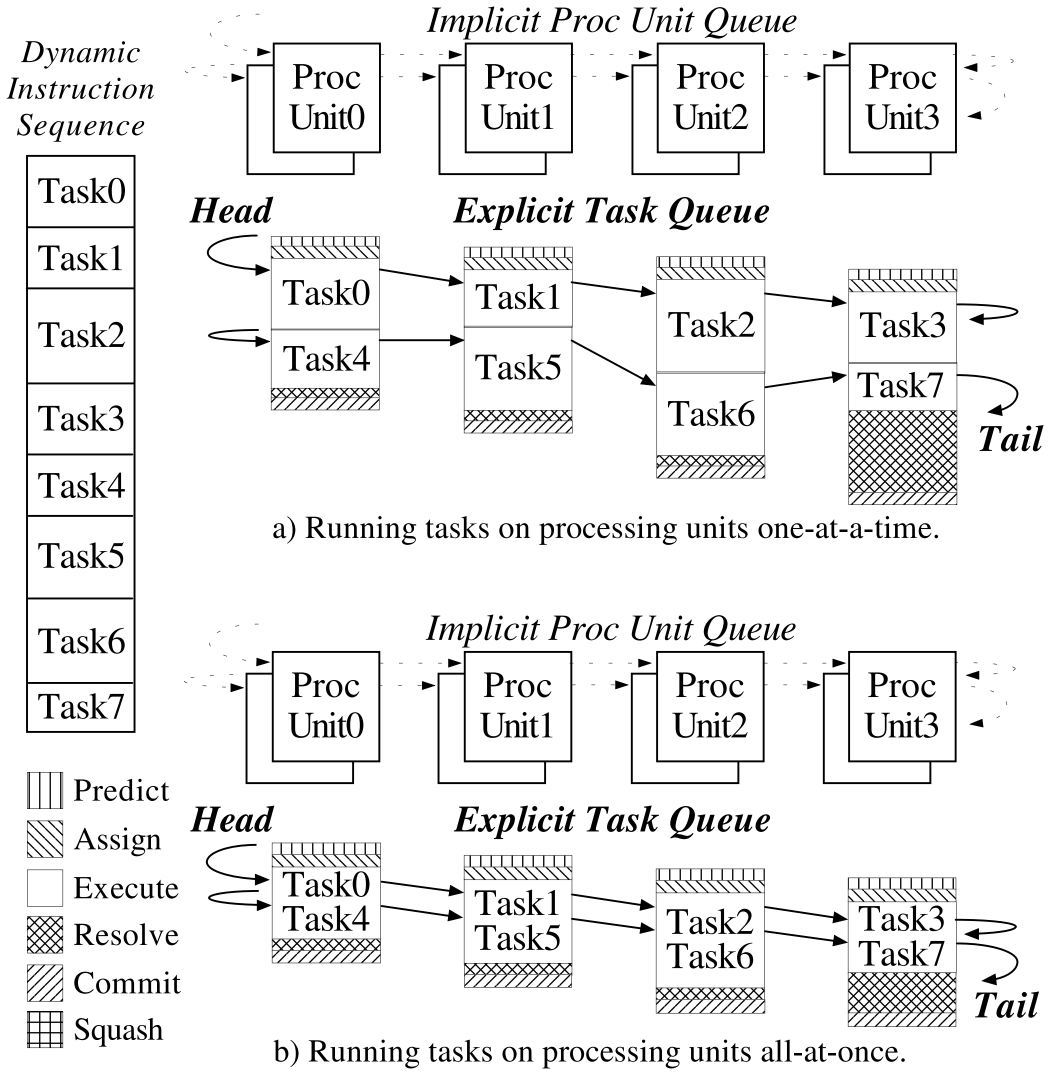


Figure 4.4: Running multiple tasks on each processing unit.

using dynamic instruction scheduling.

4.5.3.1 Dynamic Instruction Scheduling

If instruction-level parallelism exists and can be exploited within tasks, then it may serve to reduce the execution time of individual tasks and/or to shorten the critical path through a sequence of tasks. The most common means of exploiting such instruction-level parallelism in state of the art processors is dynamic instruction scheduling. The basic method by which dynamic instruction scheduling is performed involves collecting instructions into an instruction window, determining which instructions are independent, and issuing those instructions (not necessarily in program order and possibly many at a time) to functional units for execution. To what extent instruction-level parallelism may be exploited within tasks depends on the size of the instruction window that can be exposed within a task, whether the instructions of the window need not be selected in strict program order, and how many of these instructions may be issued and executed each cycle.

4.5.3.2 Microarchitecture

In light of the demands that dynamic instruction scheduling may place upon processing units, it is clear that a variety of factors must be considered in choosing their microarchitecture characteristics. As processing units are similar to conventional processors, it is natural to lever mainstream processor technology for their design. The ubiquitous superscalar processor cores in modern high performance processors are a natural choice for this purpose. Moreover, superscalar processors are a well understood (based on a body of knowledge covering the past 20 years) and commercially viable (based on the present microprocessor offerings) design strategy. This choice should not be misconstrued as a requirement, though. The multiscalar model is flexible in terms of the strategy used to perform the computation of tasks.

Nevertheless, the focus here is on the use of processing units based on superscalar techniques. Both academia [1, 55, 58, 68, 69, 81, 86, 89, 95] and industry [18, 31–36, 61, 77, 79, 82, 91–93] have considered a wide range of different superscalar designs to implement dynamic instruction scheduling. The performance of these designs as well as their costs in ideal and real implementations varies widely. Yet, because most of these techniques are applicable to the processing units of a multiscalar processor, the details of the design are not as important as the basic characteristics in terms of the size of the instruction window, the capability to select instructions, and the capacity to issue and execute multiple instructions. The design studied here is a straightforward one representative of the designs studied by academia and industry, but not based on any microarchitecture in particular.

Figure 4.5 illustrates the superscalar microarchitecture of the processing units (similar to that described in [83]). The major steps of instruction processing performed by the microarchitecture are (1) prediction and instruction memory access, (2) decode and dependence analysis, (3) dispatch and issue, (4) execution and data memory access, and (5) reorder and retire. Underlying the microarchitecture is a pipelined implementation where specific pipeline stages may or may not be aligned with the major steps of instruction processing. The simple pipeline shown in Figure 4.6 is assumed in this study. Each of the major steps described is performed in a distinct pipeline stage: **F**, **D**, **I**, **E**, **R**. Each stage takes a cycle, except for **E**, which may take multiple cycles (as indicated by the loop back above it).

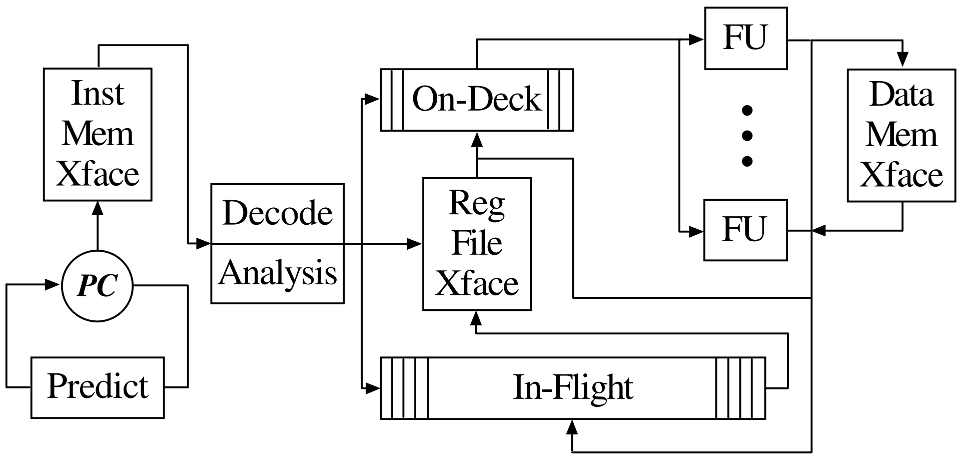


Figure 4.5: Microarchitecture of processing units.

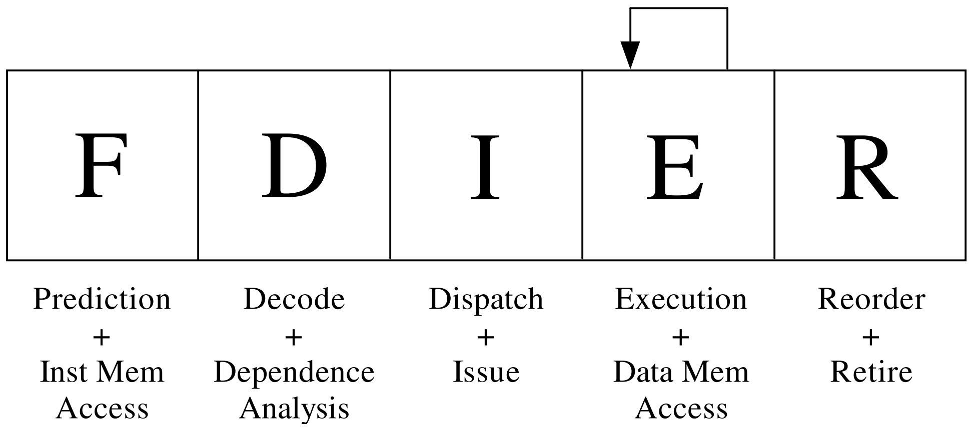


Figure 4.6: Pipeline underlying microarchitecture of processing units.

4.5.3.3 Instruction Window

The instruction window design adopted for the processing units is a fairly straightforward one. A dynamic path through a task is speculated via prediction (**Predict** in Figure 4.5). The instructions on this path are brought into the processor through the instruction memory interface (**Inst Mem Xface** in Figure 4.5). The supplied instructions are fed into the pipeline to be decoded and to have their dependences analyzed (**Decode** and **Analysis** in Figure 4.5).

From the decode and dependence analysis process, the control and data (register and memory) dependences are mapped to processor structures that ensure their proper resolution. At this point, the instructions are ready to be dispatched into the instruction window. The instruction window is handled by two principal structures: the in-flight queue and the on-deck queue. Instructions are dispatched into both queues, since each serves a different purpose.

Though most state of the art processor designs contain such structures (in some form or another), there is no consistent nomenclature for them. This work names these structures and defines their characteristics as follows. The in-flight queue (**In-Flight** in Figure 4.5) maintains the program order among instructions to allow them to be reordered and retired after execution. The on-deck queue (**On-Deck** in Figure 4.5) holds instructions until their operands are available for execution, at which point instructions may be selected and issued to the functional units for execution. The on-deck queue may be made much smaller than the in-flight queue yet still retain very high performance. The reason is that only a portion of the instructions in the machine are being scheduled; the other instructions in the machine are in various (possibly different) stages of execution. Moreover, because this structure is usually a critical timing path of a processor implementation, the smaller the queue, the faster the clock that can be supported [64].

4.5.3.4 Instruction Selection and Issue

Using the instruction window established for each task via the in-flight queue and the on-deck queue, instructions are selected and issued to the functional units for execution. There are two basic strategies to select instructions from the window: in-order or out-of-order. An in-order processor must select instructions in the order they appear in the program. An out-of-order processor may select instructions from anywhere in the instruction window subject only to the constraints of the dependences between them. In both instruction selection strategies, instructions are usually allowed to complete their execution out-of-order to accommodate different latencies for different operations. In general, out-of-order processors allow greater scheduling flexibility and thereby provide higher performance than in-order processors [38].

Among the instructions selected, it may be possible to issue many of them at the same time. The number of instructions a processor may issue each cycle is referred to as its issue width. In order to issue multiple instructions it must be ascertained that the instructions have no control, data, or structural hazards that prevent their concurrent issue. The checks that must be performed to ensure that these conditions are met increase in complexity as the issue width and the instruction window size increase [64]. In addition, the amount of register operand bandwidth that must be provided through the register file interface (**Reg File Xface** in Figure 4.5) increases as the issue width increases. As the clock cycle time may be tied to these factors, multiple instruction issue may need to be applied in a judicious manner to avoid an undue impact on the overall performance of the processor.

4.5.3.5 Instruction Execution

To support the issue of multiple instructions each cycle, the execution resources of the processor are usually divided into different functional units (**FU** in Figure 4.5) based on the type of operation performed. The functional units perform the operation specified by an instruction; a load or a store operation is performed through the data memory interface (**Data Mem Xface** in Figure 4.5). The number and type of the functional units often determines the flexibility with which scheduling decisions may be made as well as the amount of concurrency that may be exploited within the processor. Because functional units incur cost in terms of chip area and communication paths to route data to and from them, it is important to factor how many and what types are needed as part of the dynamic instruction scheduling.

4.5.3.6 Many-To-One Impact

The impact of the many-to-one assignment of tasks to processing units need not be a fundamental change in the processing unit microarchitecture described thus far. The principal change is that some processing unit resources may need to be duplicated. (Other resources in the instruction supply and data supply may need to be duplicated as well, but these are not discussed here.) The extent to which these resources need to be duplicated depends on the running policy used for multithreading. For the conservative one-at-a-time policy described above, only duplicate program counters are needed. The rest of the processing unit need not be changed since each task executes one-at-a-time and does not share resources with other tasks. On the other hand, for the aggressive all-at-once policy described above, duplicate in-flight and on-deck queues are provided in addition to duplicate program counters to differentiate between the instruction windows of each task. The instructions issued may be taken selected from any combination of the instruction windows, though priority is given to older tasks over newer tasks. Because the instructions do share resources, each instruction is given a unique task tag to identify the instruction window to which it belongs.

4.6 Processing Unit Evaluation

The processing unit evaluation is divided into three parts to better understand the individual as well as the collective influence of processing unit design factors.

First, an inter-task study is performed that focuses on the extent to which inter-task instruction-level parallelism is extracted by different processing unit designs. For this inter-task study, the evaluation involves varying the number of tasks and the number of processing units to explore the range of performance delivered by the straightforward one-to-one and the more involved many-to-one task to processing unit assignment strategies.

Second, an intra-task study is performed that focuses on the extent to which intra-task instruction-level parallelism is extracted by different processing unit designs. For this intra-task study, the evaluation involves varying the characteristics of the processing units to explore the range of performance delivered by different instruction window, instruction selection and issue, and instruction execution configurations.

Finally, the processing unit designs from the inter-task and intra-task studies are put together to focus on the extent to which instruction-level parallelism is extracted overall by different processing unit designs. To study the overall performance, the evaluation involves varying the characteristics of the processing units as well as the numbers of tasks to investigate how different inter-task/inter-task configurations extract different amounts of instruction-level parallelism from the dynamic instruction sequence.

4.6.1 Metrics

The fundamental metric with which to evaluate processing performance is time. For the framework in which this evaluation is performed, time must be measured in terms of number of cycles, rather than elapsed wall-clock time because the actual cycle time in not modeled in the simulator used for these studies. Yet, the number of cycles as measured by the simulator is a somewhat inconvenient metric because many benchmarks are used for the evaluation, each with its own input that runs for a different number of instructions. Moreover, the number of cycles does not provide a direct indication of the amount of instruction-level parallel execution achieved by a particular configuration under study.

A more intuitive metric that correlates with performance as a result of instruction-level parallel execution is the number of useful instructions completed per cycle (or just instructions per cycle). Since a particular benchmark is always run for the same number of instructions (as a result of using the same benchmark binary and the same benchmark input), the instructions per cycle is an appropriate metric for the performance. Each benchmark is simulated for a complete run with its specified input to provide the steady state instructions per cycle. With so many benchmarks, though, it is difficult to identify performance trends looking at each of the individual benchmarks.

Therefore, this evaluation provides the performance in terms of instructions per cycle as the unweighted harmonic mean (HMEAN) for each of the two groups of benchmarks, SPEC CINT95 and SPEC CFP95, respectively. Though each benchmark executes for a different number of instructions, the instructions per cycle should not be weighted because each benchmark program is meant to be representative of a particular type of (integer or floating point) application independent of the number of instructions it executes for its particular input. Moreover, the harmonic mean (HMEAN) is the correct mean (as opposed to the arithmetic or geometric) because instructions per cycle is inversely proportional to program execution time.

4.6.2 Inter-Task Study

In order to isolate the effect of inter-task instruction-level parallelism in a multiscalar processor, the processing units used for this part of the evaluation are fixed with 4-wide out-of-order characteristics. With these processing unit characteristics, the task assignment (one-to-one and many-to-one) and task mapping (one-at-a-time and all-at-once) policies for these processing units are varied among those described earlier. In addition, since only the processing units are studied here, the other multiscalar processor components are fixed to the standard configuration as given in Table 3.11 of Chapter 3.

4.6.2.1 One-To-One Assignment

The straightforward task to processing unit assignment strategy (discussed in the earlier text on processing unit design) assigns tasks to processing units one-to-one. Figures 4.7 and 4.8 show the performance obtained for each of the SPEC CINT95 and SPEC CFP95 benchmarks as the total number of tasks and likewise (since the assignment is one-to-one) the total number of processing units is increased.

The thick dashed line in the figures indicates the harmonic mean of the instructions per cycle for all of the benchmarks, and the thin lines (each with a unique marker) in the figures indicate the instructions per cycle for each individual benchmark. (Throughout the rest of this thesis only the means for the integer and floating point benchmarks are reported in order to make it easier to identify the performance trends without the distraction of individual benchmark differences.)

Considering all of the figures, notice that as the number of tasks is doubled, performance grows steadily for all of the integer programs and for all of the floating point programs. Nonetheless, it is particularly significant that performance is only increasing at a linear rate, while the cost in terms of the number of processing units to deliver this performance is increasing at an exponential rate.

As shown in Table 4.1, the overall performance of the processor for one-to-one assignment indeed increases with more processing units, but the performance per processing unit decreases. The information in the table confirms the concerns/challenges postulated earlier with respect to instruction-level parallelism. That is, high performance depends on exposing instruction-level parallelism with many tasks, but exploiting it via one-to-one assignment incurs high cost but low return per processing unit.

One-To-One Proc Unit Config	HMEAN Insts Per Cycle			
	SPECINT95		SPECFP95	
	Overall	Per PU	Overall	Per PU
1	1.25	1.25	1.36	1.36
2	1.60	0.80	2.11	1.05
4	2.28	0.57	3.33	0.83
8	2.93	0.37	4.84	0.60
16	3.36	0.21	6.09	0.38

Table 4.1: Overall performance and performance per processing unit for one-to-one assignment (running a single task per processing unit, with the total number of tasks equal to the number of processing units).

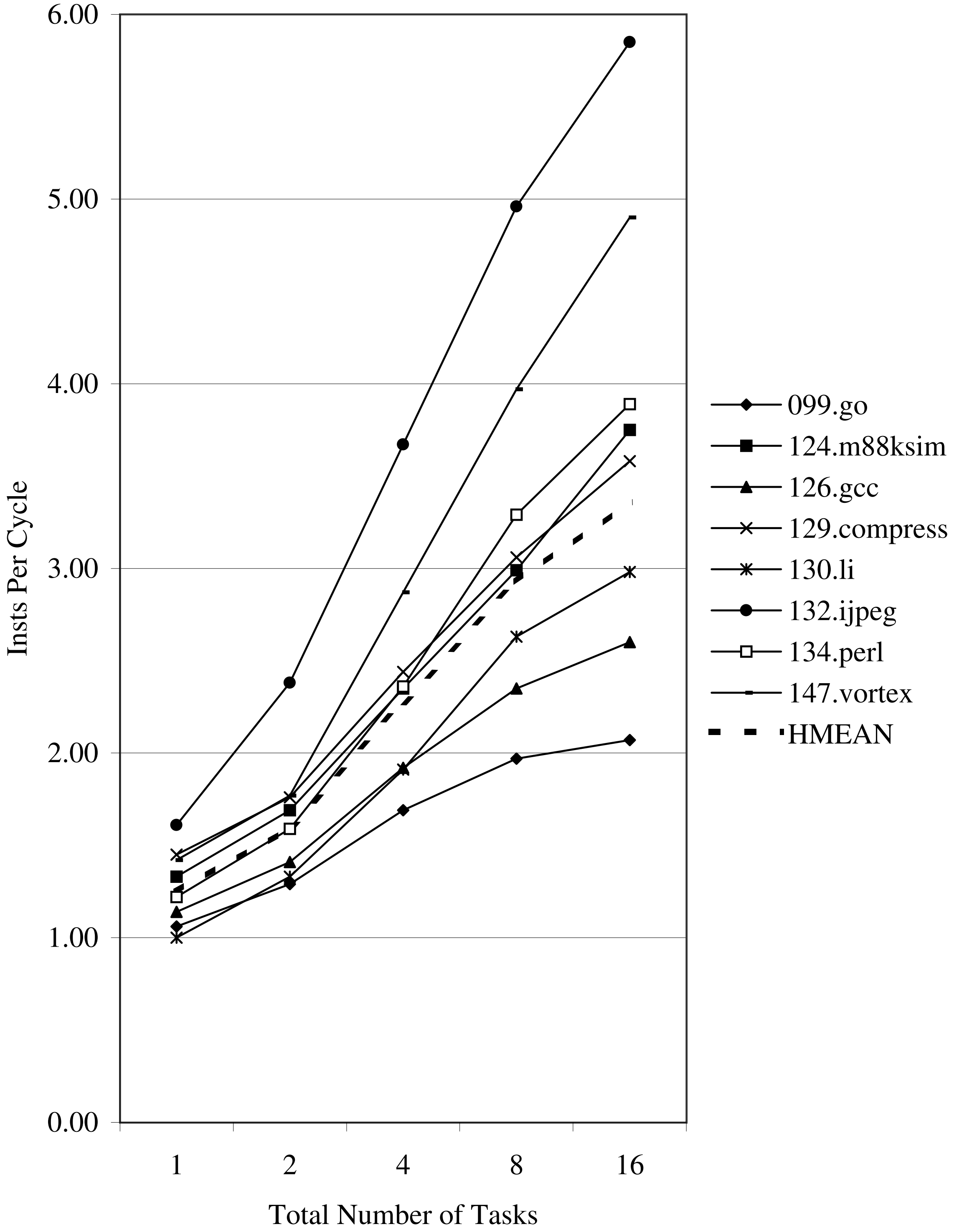


Figure 4.7: Growth in IPC as a function of tasks for SPEC CINT95 benchmarks.

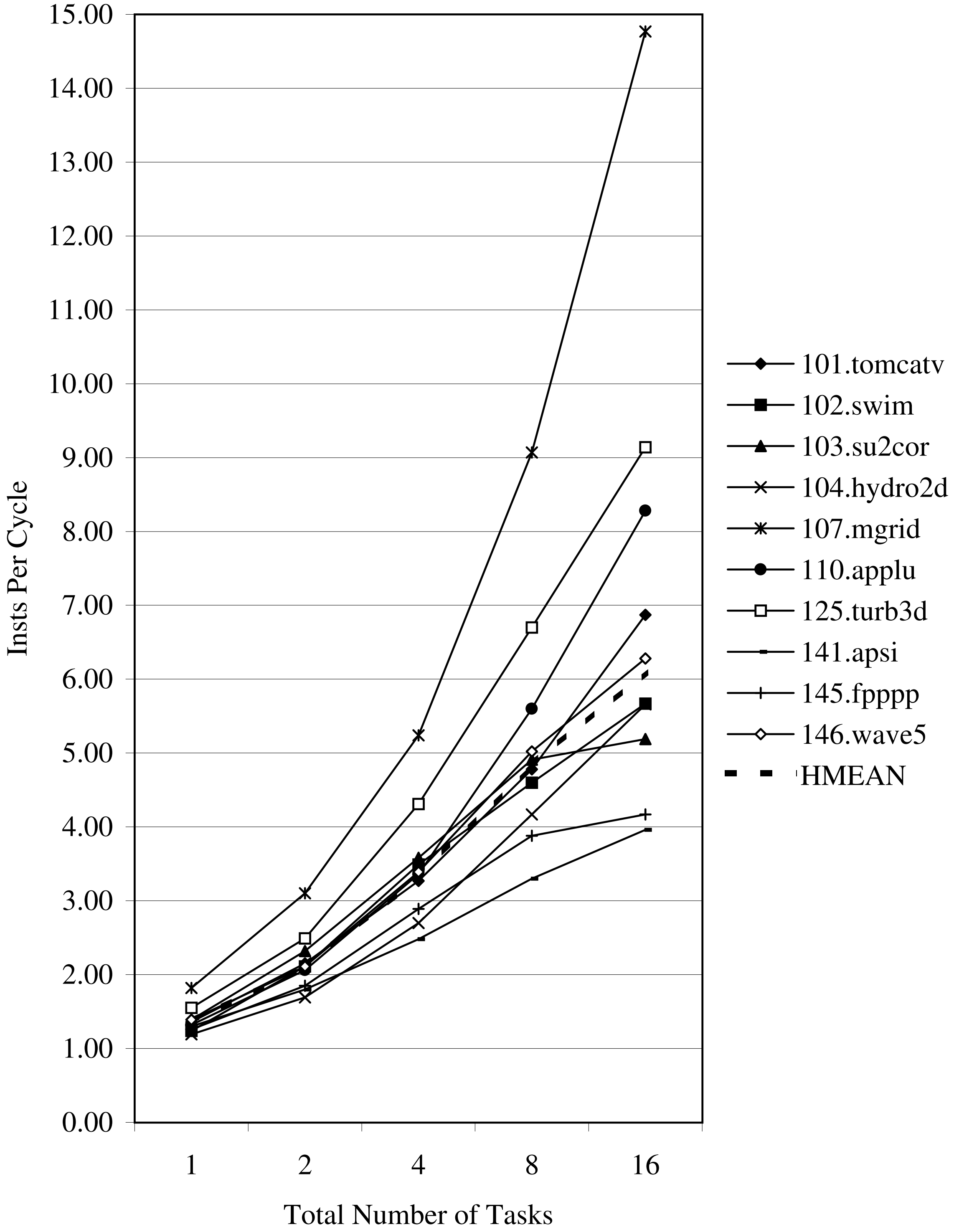


Figure 4.8: Growth in IPC as a function of tasks for SPEC CFP95 benchmarks.

4.6.2.2 Many-To-One Assignment

The alternative task to processing unit assignment discussed in the processing unit design assigns tasks to processing units many-to-one in order to address the cost-performance drawbacks of the one-to-one assignment. The processing units are of exactly the some configuration as before, except for modifications necessary to support the many-to-one multithreading mechanism as well as the one-at-a-time and all-at-once running policies. Only the round-robin mapping policy is used for this study. Figures 4.9 and 4.10 show the performance obtained for each of the SPEC CINT95 and SPEC CFP95 benchmarks as the total number of tasks is increased. However, in contrast to the earlier figures (Figures 4.7 and 4.8), there are multiple harmonic mean lines, each of which represents a configuration of the processing units where the number of processing units for each line remains fixed, but the number of tasks per processing unit is increased (and the total number of tasks) moving left to right along the line in graph. The dashed lines indicate processing units configured with the one-at-a-time running policy. The solid lines indicate processing units configured with the all-at-once running policy.

Notice in the figures, that performance increases as the number of tasks per processing unit (though not the number of processing units) is doubled. However, for the one-at-a-time running policy, the improvement seems to fall off much past two tasks assigned per processing unit. For instance, looking at the dashed line in the 2.one case, the curve stops rising above 4 total tasks, which is 2 tasks per processing unit; for the dashed line in the 4.one case, the curve stops rising above 8 total tasks, which again is 2 tasks per processing unit. For the all-at-once running policy, the improvement is linear or nearly linear, though it too seems to fall off, but past 4 tasks assigned per processing unit. For instance, looking at the solid line in the 1.all case, the curve stops rising above 4 total tasks, which is 4 tasks per processing unit; likewise, for the solid line in the 2.all case, the curve stops rising above 8 totals tasks, which as before is 4 tasks per processing unit.

Moreover, the all-at-once running policy outperforms the one-at-a-time running policy in general. Furthermore, the gap between the two policies increases as the number of tasks assigned per processing unit increases (more pronounced for floating point than for integer benchmarks). With a fixed number of tasks, in particular 16 tasks as shown in Table 4.2, the overall performance of the processor using many-to-one assignment increases with more processing units, and though the performance per processing unit still decreases, the value is improved over the case for the one-to-one assignment shown in Table 4.1 (by a smaller amount for integer programs and a larger amount for floating point programs). Note that the 16 processing unit performance is the same in both tables since with only 16 tasks each processing unit is only assigned 1 task in both cases. The 2, 4, and 8 processing unit performance is different in the many-to-one assignment table than in the one-to-one assignment table because processing units are only assigned a single task in the one-to-one case, but are assigned multiple tasks in the many-to-one case.

An important question to consider is how does many-to-one assignment using some number of processing units, compare to one-to-one assignment using more, less, or the same number of processing units. One way to find an answer, for the particularly interesting case when each runs the same number of tasks, is to look in the Figures 4.9 (for integer) and 4.10

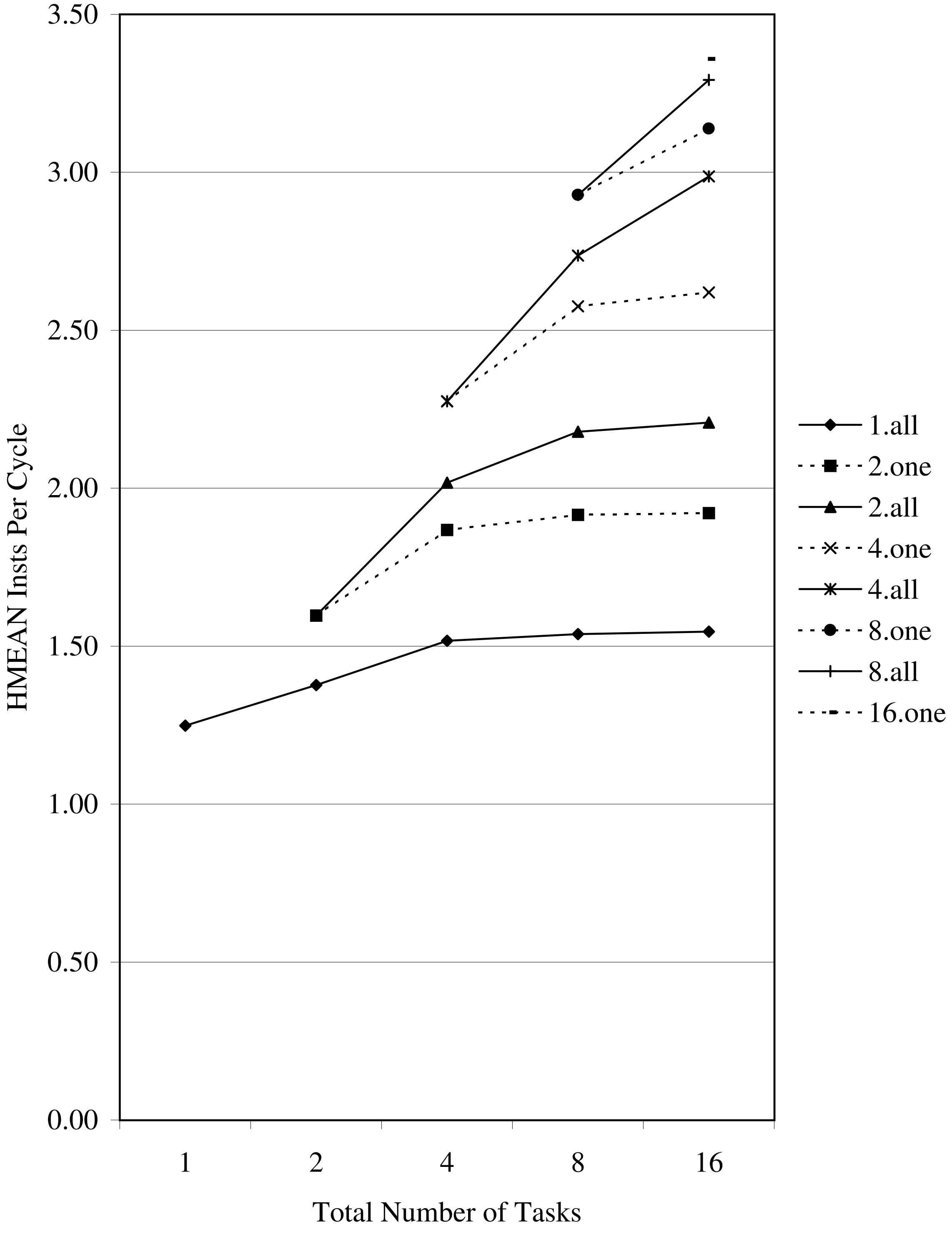


Figure 4.9: Harmonic mean of IPC for SPEC CINT95 benchmarks.

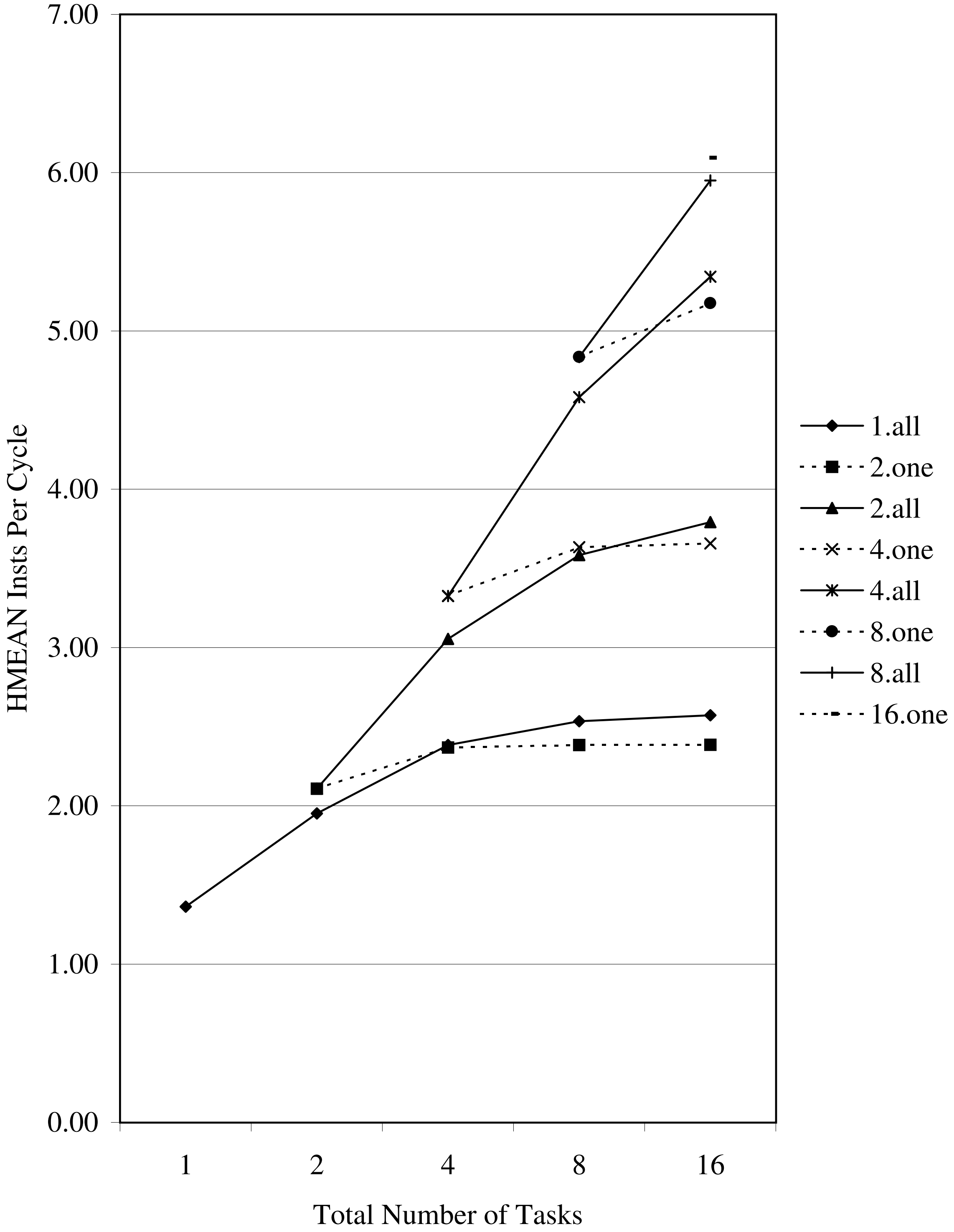


Figure 4.10: Harmonic mean of IPC for SPEC CFP95 benchmarks.

Many-To-One Proc Unit Config	HMEAN Insts Per Cycle			
	SPECINT95		SPECFP95	
	Overall	Per PU	Overall	Per PU
1.all	1.55	1.55	2.57	2.57
2.one	1.92	0.96	2.39	1.19
2.all	2.21	1.10	3.79	1.90
4.one	2.62	0.65	3.66	0.91
4.all	2.99	0.75	5.34	1.34
8.one	3.14	0.39	5.17	0.65
8.all	3.29	0.41	5.95	0.74
16.one	3.36	0.21	6.09	0.38

Table 4.2: Overall performance and performance per processing unit for many-to-one assignment (running multiple tasks per processing unit, with a total number of 16 tasks distributed equally across the processing units).

(for floating point) at a vertical slice for a particular number of tasks (indicated at the bottom of the vertical slice). The top marker of the vertical slice indicates the performance for the one-to-one assignment of tasks to processing units. For a fixed number of tasks, the highest performance is attained when each task runs on its own processing unit. However, by moving down the vertical slice and examining other marks, it may be seen what level of performance may be attained using an equal number of tasks, but using fewer processing units running them one-at-a-time or all-at-once (according to the line with which the marker is associated).

The markers beneath the top marker indicate the performance for different many-to-one assignment configurations. Consider the vertical slices in the integer and floating point figures associated with 16 tasks. For the integer benchmarks, performance increases as the running policy is changed from one-at-a-time to all-at-once (*e.g.* the 4.all line is above the 4.one line), and as the number of processing units is increased (*e.g.* the 8.one line is above the 4.one, and the 8.all line is above 4.all line). For the floating point benchmarks, performance increases as the number of processing units is increased (*e.g.* the 8.one line is above the 4.one, and the 8.all line is above 4.all line), but it requires twice as many processing units running one-at-a-time to attain nearly (but not quite) the same performance as for all-at-once (*e.g.* the 8.one line is slightly below the 4.all line).

4.6.3 Intra-Task Study

In the same way as for the inter-task study, the effect of intra-task instruction-level parallelism is isolated from that of inter-task for this part of the study. In order to do so, the task assignment policy is fixed as one-to-one, and the task running policy is fixed as one-at-a-time (for a single task per processing unit). The characteristics of the processing units are varied to reflect the aspects of dynamic instruction scheduling being studied. The rest of the multiscalar processor components are maintained in the standard configuration shown in Table 3.11 of Chapter 3.

4.6.3.1 Instruction Window

The design of the instruction window in relation to dynamic instruction scheduling may have a significant impact on the amount of instruction-level parallelism that may be extracted as well as the clock frequency of the processor. In particular, there is usually a trade-off between the instruction window size and the clock frequency of its design; the smaller the window, the faster the clock, and the lower the parallelism; the larger the window, the slower the clock, and the higher the parallelism. However, a multiscalar processor does not extract instruction-level parallelism in exactly the same as a superscalar processor, so performance is not as sensitive to the window size.

The performance of a multiscalar processor is influenced primarily by the size of the collective instruction window provided by all of its processing units, and only secondarily by the size of the instruction window for each individual processing unit. That is, the instruction window of each processing unit in a multiscalar processor need only support the execution of a task, so it may not need to be as big as that of a superscalar processor in order to achieve most of its performance potential. Based on much empirical observation, the size of the in-flight queue need be no bigger than the average task size, and for a particular in-flight queue size, the on-deck queue need be no bigger than half the in-flight queue size.

Figures 4.11 and 4.12 show the harmonic mean of the instructions per cycle of the SPEC CINT95 and SPEC CFP95 benchmarks for a range of individual processing unit window configurations. The performance is reported as a stacked bar for multiscalar processors using different number of processing units (1, 2, 4, 8, and 16 here). The number of processing units for the processor is varied to indicate what (if any) effect this factor has on the performance with different processing unit window configurations. That is, the focus is on individual processing unit window configurations with different in-flight queue and on-deck queue characteristics, given as **F**x**D**, where **F** is the in-flight queue size and **D** is the on-deck queue size.

Consider the in-flight queue. For the integer benchmarks, there is little effect from using larger in-flight queues. The reason for this behavior is that the integer tasks are relatively small and fit within a small in-flight queue. Thus, there is nothing to be gained by increasing the size of the in-flight queue. However, for the floating point benchmarks, there is modest improvement from using larger in-flight queues. Because the floating point tasks are relatively big, the larger in-flight queue allows the processing unit to expose more instructions from a task and thereby provides more opportunities to exploit instruction-level parallelism within a task.

Consider the on-deck queue. For the integer benchmarks, there is some effect from using larger on-deck queues. If the on-deck queue is very small, only a few instructions, then it is easily clogged by only a few dependent instructions which stall issue. Even though integer tasks are relatively small, the instructions within the tasks may be quite dependent. Likewise,

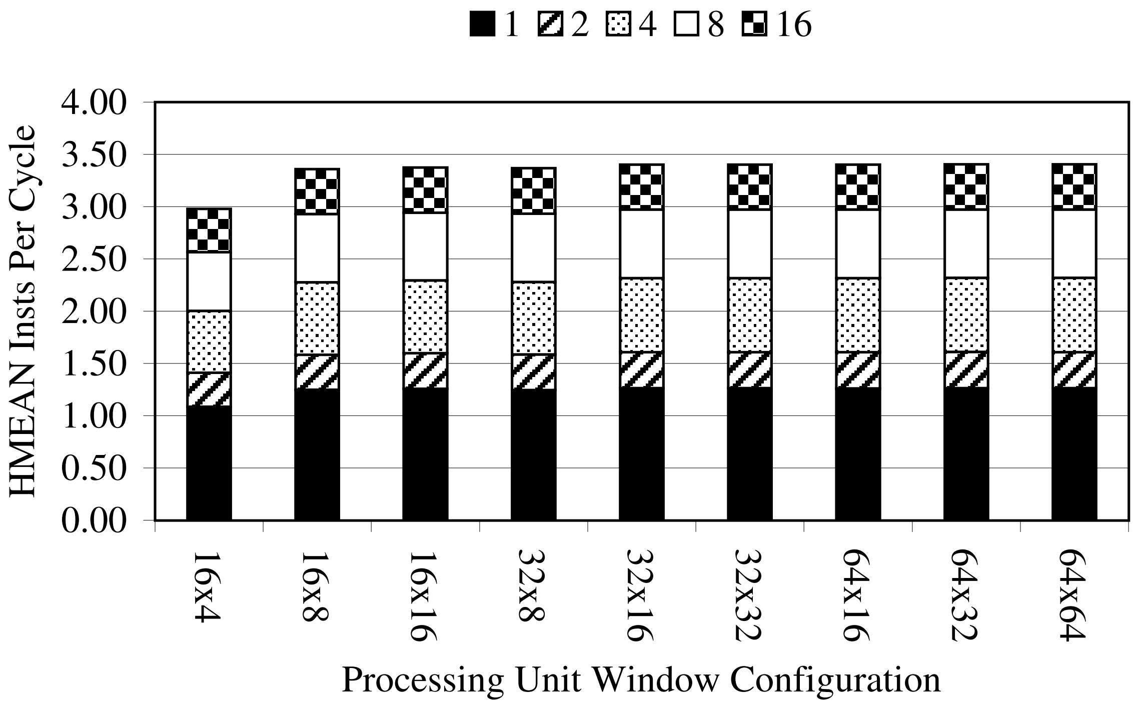


Figure 4.11: Harmonic mean of IPC for SPEC CINT95 benchmarks.

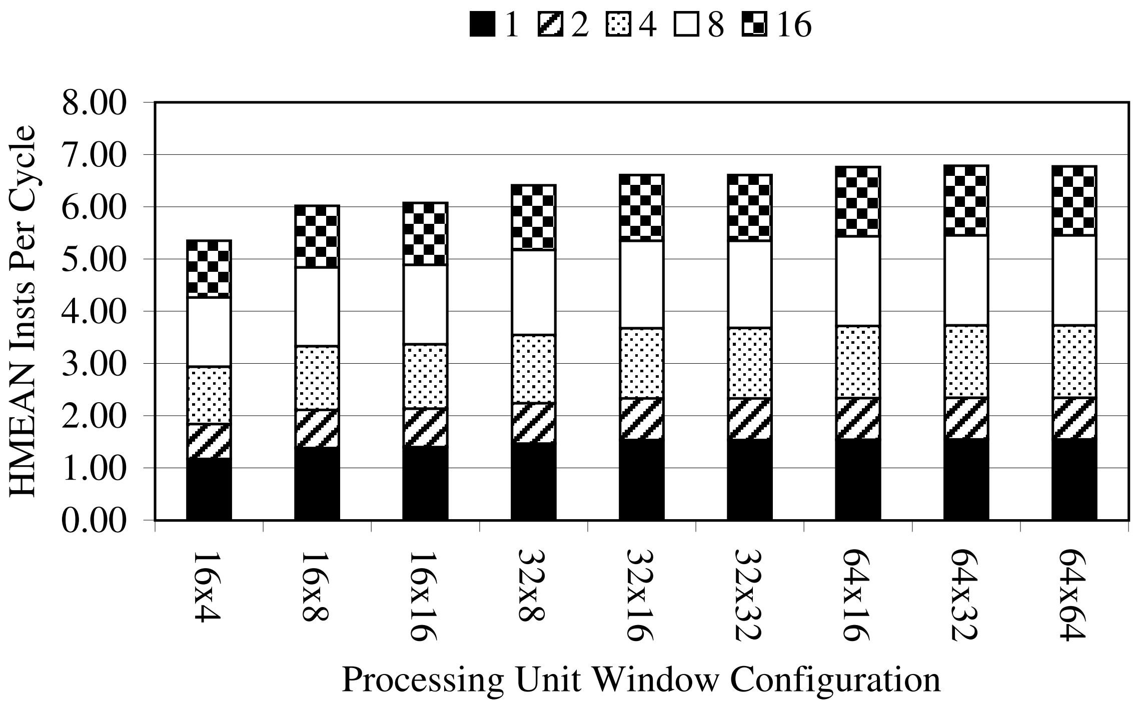


Figure 4.12: Harmonic mean of IPC for SPEC CFP95 benchmarks.

for the floating point benchmarks, there is some effect from using larger on-deck queues. Though the instructions within floating point tasks may not be as dependent as those in integer tasks, the floating point operations are usually of longer latency than integer operations. As a result, instruction issue may be prone to clog because dependent instructions tend to remain in the on-deck queue longer.

Overall, there is not a significant advantage in using bigger in-flight queues and/or on-deck queues for the instruction window over small ones. Since integer tasks are small, the size of the window is not of much concern. However, since floating point tasks are big, the window size is somewhat of a performance factor, but not a critical one. In this work, the multiscalar designs focus on windows with fairly small in-flight and on-deck queues since the benefits of supporting a fast clock are assumed to outweigh the small improvements in instruction-level parallelism afforded by the larger in-flight and on-deck queues.

4.6.3.2 Instruction Selection and Issue

The key characteristics that are usually the focus with respect to dynamic instruction scheduling are the number of instructions that may be issued at the same time and whether these instructions are selected from the instruction window in-order or out-of-order. The characteristics of the processing units are varied from in-order to out-of-order selection using uniform (any type of instruction) issue of 1, 2, or 4 instructions each cycle. In addition, it is important to focus on the effects of these two aspects in relation to the total number of processing units that make up the multiscalar processor. Thus, each processing unit configuration is evaluated in a multiscalar processor of 1, 2, 4, 8, and 16 processing units. The results of these configurations are shown in Figures 4.13 and 4.14 for the SPEC CINT95 and the SPEC CFP95 benchmarks, given as **S**.u**I**, where **S** is **io** for in-order or **oo** for out-of-order selection, and **I** is a uniform issue width of **1**, **2**, or **4** instructions.

The overall performance in terms of instructions per cycle increases as the multiple issue capabilities of the processing units are increased from 1 to 2 to 4 issue for both in-order and out-of-order selection designs. The out-of-order designs outperform the in-order designs for equal issue width in all cases. If the instruction selection and issue capabilities are considered together, it can be seen that there is a steady increase in performance going from left to right in the figures. However, the rate of performance increases is higher as the issue width is increased for out-of-order versus in-order processing units. As a result, the 2-wide out-of-order and the 4-wide in-order represent what appears to be a break even point, where it might be more advantageous to use out-of-order selection than to increase multiple issue for in-order selection. Moreover, this effect on overall performance is amplified as the number of processing units increases.

4.6.3.3 Instruction Execution

The instruction selection and issue capabilities described above assume uniform issue across the functional units of the processing unit. However, most traditional superscalar processors are not designed with uniform issue characteristics. Instead the processor has a collection of functional units where often no single type of operation can take advantage of the entire issue

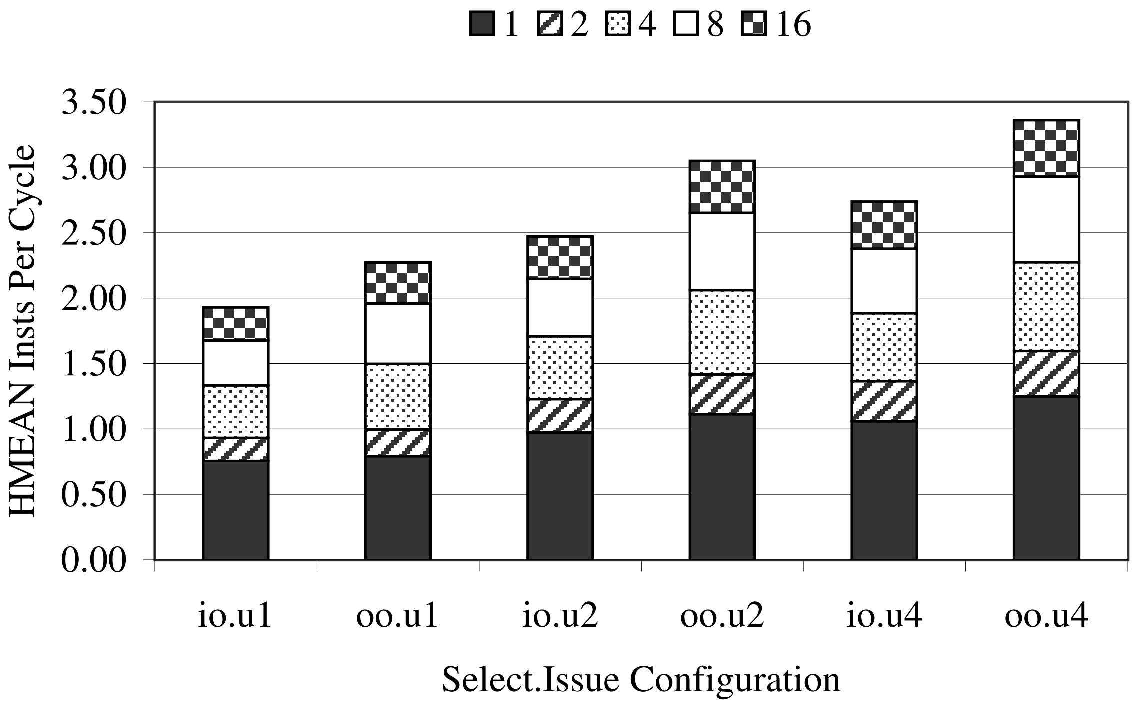


Figure 4.13: Harmonic mean of IPC for SPEC CINT95 benchmarks.

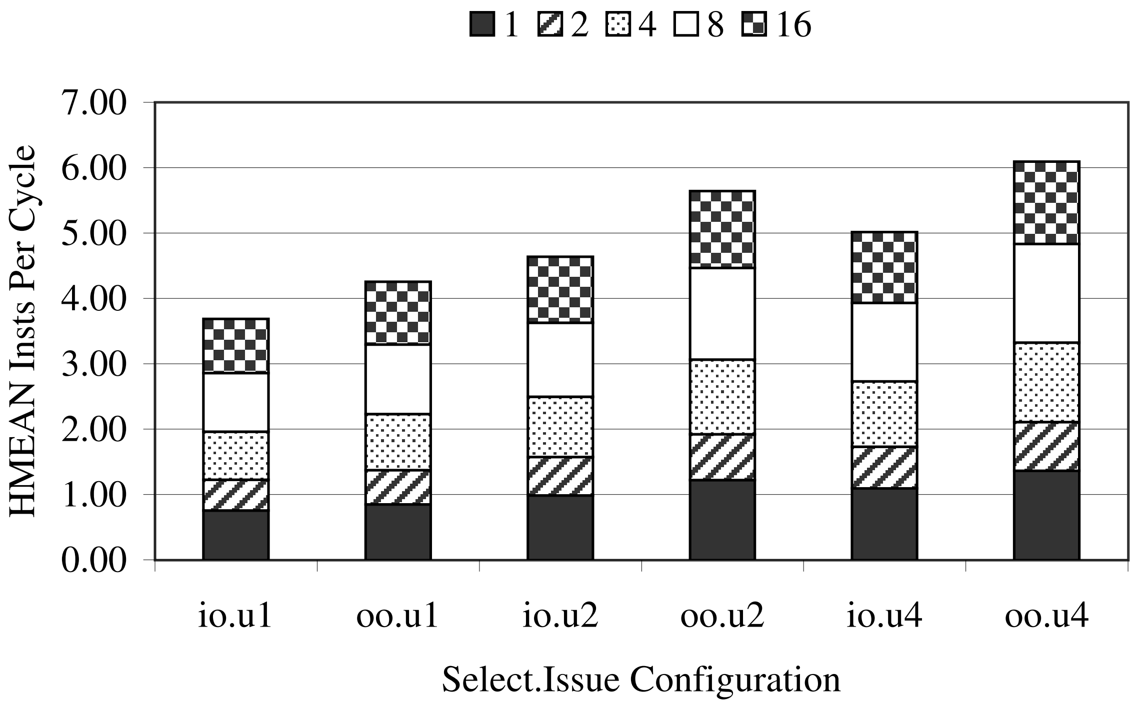


Figure 4.14: Harmonic mean of IPC for SPEC CFP95 benchmarks.

width of the processor. To focus on how this factor impacts performance, the functional units may be classified into three broad categories: compute, memory, and control. The compute functional units are those that perform any integer or floating point arithmetic operations. The memory functional units are those that perform loads and stores. The control functional units are those that perform all branches and procedure calls. This classification is more general than that of an actual processor where compute functional units consist of different types for different arithmetic operations, for integer and floating point operations at the very least. Nevertheless this classification is specific enough that the mix of these functional units may be varied to see the impact on overall performance.

The configurations of the functional units studied are given in Table 4.3. For these configurations, there are always at least as many compute functional units as there are memory functional units. Though it is relatively simple to provide more compute functional units, it is relatively complex to provide more memory functional units. In an actual design, the difficulty of dynamic memory disambiguation and the need for more ports on the memory storage to be accessed usually incurs significant cost. Therefore, in terms of cost-performance compute functional units are cheap and memory functional units are expensive. The control functional units are not really given much concern in these configurations.

Config	Issue Width	Number of Functional Units		
		Category		
		Compute	Memory	Control
u1	1	1	1	1
h2	2	1	1	1
uh2	2	2	1	1
u2	2	2	2	2
q4	4	1	1	1
hq4	4	2	1	1
h4	4	2	2	1
uq4	4	4	1	1
uh4	4	4	2	1
u4	4	4	4	4

Table 4.3: Configurations of functional units.

One control functional unit suffices in this study because both integer and floating point tasks contain relatively few control flow instructions. Only the uniform configurations have more than 1 control functional unit. In varying the number of functional units of different types, the goal is to observe how much of the instruction execution must be devoted to any particular functional unit category to achieve performance comparable to the uniform case. For all of the configurations, the processing units are assumed to be of an out-of-order design since these give a broader range of performance for different issue widths. The relative results are similar if in-order designs are used.

The results in Figures 4.15 and 4.16 show the harmonic means of the instructions per cycle for SPEC CINT95 and SPEC CFP95 benchmarks with the configurations given in Table 4.3. Each configuration is evaluated in a multiscalar processor of 1, 2, 4, 8, and 16 processing units. The uniform 1-wide issue results may be used as a baseline for comparison. The trends observed in these results apply to the integer and floating point benchmarks alike. The most significant observation to be made from the data is that the provision of at least 2 memory functional units per processing unit are needed to attain the best performance. All 2-wide issue configurations perform appreciably below uniform 2-wide because none have 2 memory functional units. Only the 4-wide configurations with 2 memory functional units perform as well as uniform, though the 4 memory functional units provided by uniform 4-wide are more than is needed. For the compute functional units, the results indicate that there is at least some improvement in providing a number of compute functional units equal to at least half of the issue width; however, providing more than half give no appreciable performance improvement. Overall, the results indicate that there is a significant increase in going form 1-wide to 2-wide issue, but only a marginal increase in going from 2-wide to 4-wide issue.

4.6.4 Putting It All Together

This section combines both the inter-task and intra-tasks aspects of processing units to consider what performance may be achieved with different configurations. These results are presented in terms of the number of tasks that are supported by a processor configuration. In particular, the break down divides the configurations into those that support 4 tasks, 8 tasks, or 16 tasks. The break down in terms of tasks is a matter of presentation convenience because there are so many configurations to consider. It is possible to compare results for configurations supporting a different number of tasks by comparing across the figures. Nevertheless, it may be difficult to draw definitive performance conclusions about different configurations with so many parameters changing at the same time.

The results are shown in the collection of figures as follows. Figures 4.17 and 4.18 show the results for configurations of 4 tasks on the SPEC CINT95 and SPEC CFP95 benchmarks, respectively. Figures 4.19 and 4.20 show the results for configurations of 8 tasks on the SPEC CINT95 and SPEC CFP95 benchmarks, respectively. Figures 4.21 and 4.22 show the results for configurations of 16 tasks on the SPEC CINT95 and SPEC CFP95 benchmarks, respectively. The results are based on runs with processing units in the standard configuration summarized in Table 3.11 of Chapter 3. However, the characteristics of instruction selection and issue are varied to reflect different processing unit designs. To be specific, both in-order and out-of-order instruction selection coupled with uniform 1-wide, 2-wide, or 4-wide instruction issue is studied.

The y-axis of the figures represents the harmonic mean of instructions per cycle. The x-axis of the figures represents the overall instruction selection and issue characteristics of the processor, given as **O**.**S**, where **O** is the overall number of instructions that may be issued (per processing unit issue width multiplied by the number of processing units), and **S** is whether

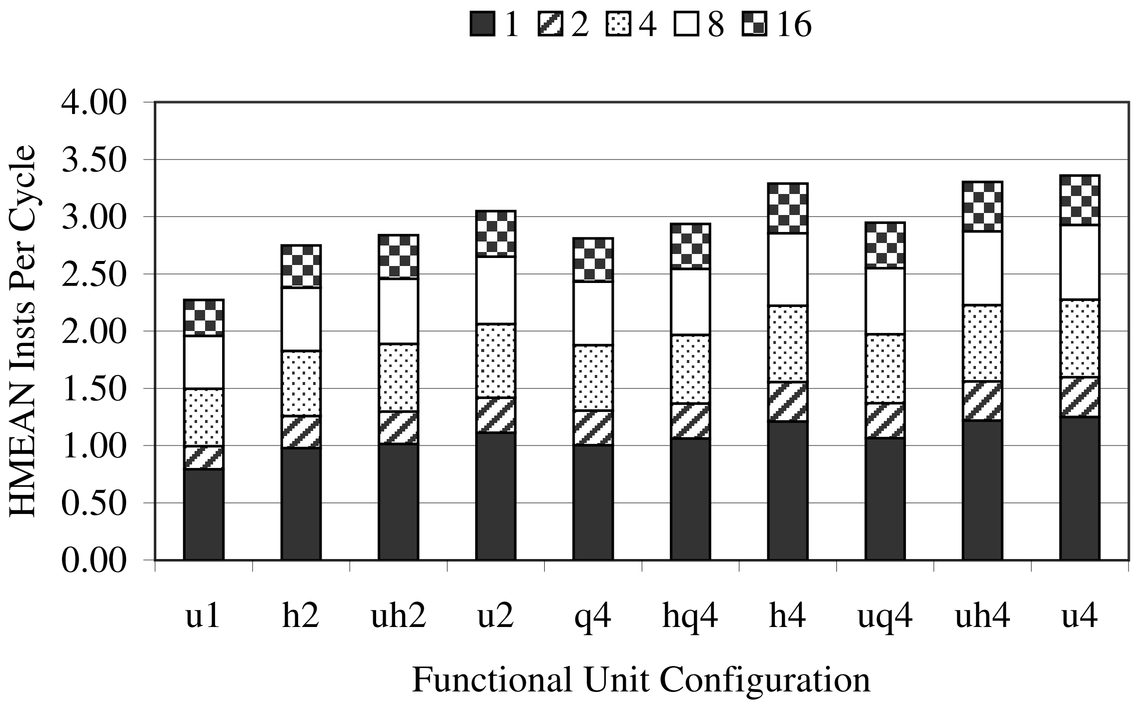


Figure 4.15: Harmonic mean of IPC for SPEC CINT95 benchmarks.

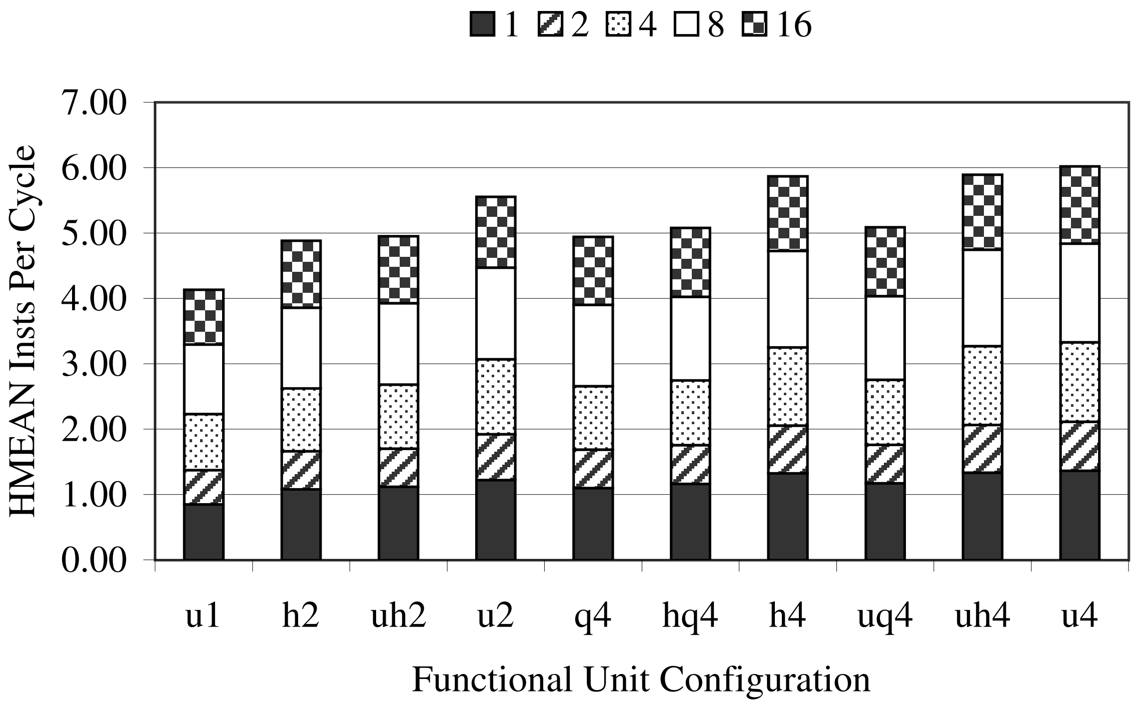


Figure 4.16: Harmonic mean of IPC for SPEC CFP95 benchmarks.

instructions are selected in-order, **io**, or out-of-order, **oo** by the processing units. Each line on the graphs is given an identifier of **N**x**T**.**run**, where **N** is the number of processing units, **T** is the number of tasks run on each processing units, and **run** is the policy by which tasks are run on the processing units, **one** for one-at-a-time or **all** for all-at-once. The use of a dashed line indicates that the processing units run tasks one-at-a-time; the use of a solid line indicates that the processing units run tasks all-at-once. Though the figures show the results for all runs to give the full perspective on performance, the discussion that follows is directed toward the middle of the overall issue characteristics represented, not the low end or the high end.

4.6.4.1 Same Number of Tasks

It is straightforward to consider processor configurations that support the same number of tasks. In comparing in-order overall characteristics with out-of-order overall characteristics, out-of-order outperforms in-order for the same issue width. Usually, an out-of-order configuration outperforms an in-order configuration with 2 times the issue width, and sometimes 4 times the issue width. For instance, in Figure 4.21, 4x4.one for 8.oo overall issue characteristics outperforms the 8x2.all for 16.io overall issue characteristics as well as 16x1.one for 32.io overall issue characteristics. Moreover, the trends indicate that, for the same overall issue characteristics, a configuration that supports many-to-one, running all-at-once is superior to a configuration that supports many-to-one, running one-at-a-time; which in turn is superior to a configuration that supports one-to-one, running one-at-a-time.

In comparing the configurations with the same overall issue characteristics that support many-to-one, running all-at-once, it is usually the case that 4 tasks per processing unit outperforms 2 tasks per processing unit for lower overall issue characteristics. For example, in Figure 4.22, 4x4.all significantly outperforms 8x2.all for 8.oo overall issue characteristics. On the other hand, 2 tasks per processing unit outperforms 4 tasks per processing unit for high overall issue. For example, in Figure 4.22, 8x2.all performs slightly better than 4x4.all for 16.oo overall issue characteristics. To some extent, 8 tasks per processing unit exhibits similar behavior with respect to 4 tasks per processing unit, but in general the difference between the two is less significant for low overall issue characteristics and more significant for high overall issue characteristics. In general, 16 tasks per processing unit performs below other many-to-one, running all-at-once configurations.

In comparing the configurations with the same overall issue characteristics that support many-to-one, running one-at-a-time, there seems to be a balancing act between the number of processing units and the number of tasks per processing unit. Usually, the advantage of mapping many tasks per processing unit diminishes as the overall issue width increases. For instance, in Figure 4.19, 4x2.one clearly outperforms 8x1.one for 8.io overall issue characteristics but performs slightly worse for 16.oo overall issue characteristics. The crossover points seem to favor more processing units over more tasks per processing unit once the issue width of the processing units is above 1-wide issue. Furthermore, the differences seem to be more pronounced for the floating point benchmarks than for the integer benchmarks. In general, the configurations that support one-to-one, running one-at-a-time seem to perform poorly on the integer programs but are fairly competitive on the floating point programs for the same overall issue characteristics.

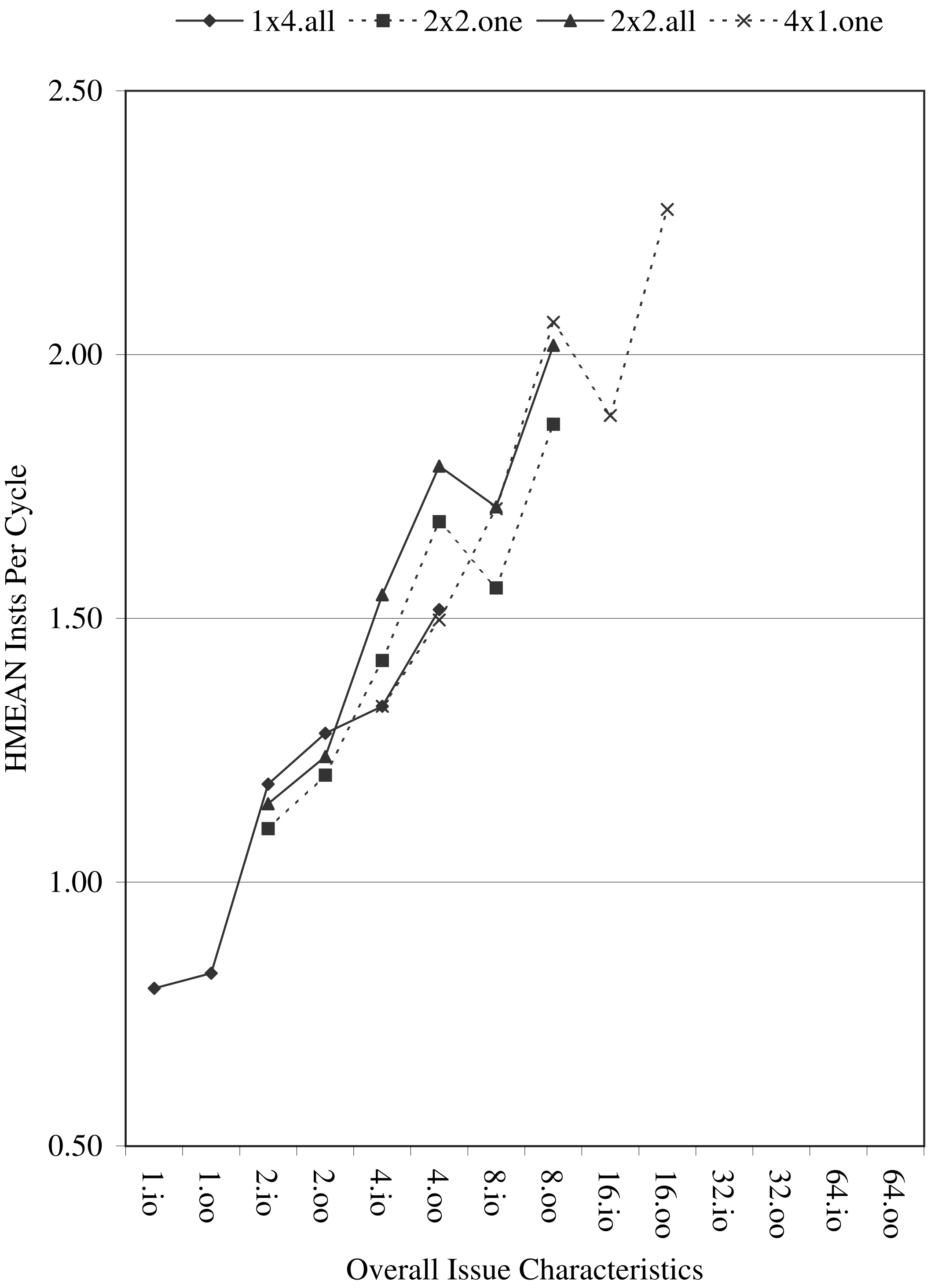


Figure 4.17: Harmonic mean of IPC using 4 tasks for SPEC CINT95 benchmarks.

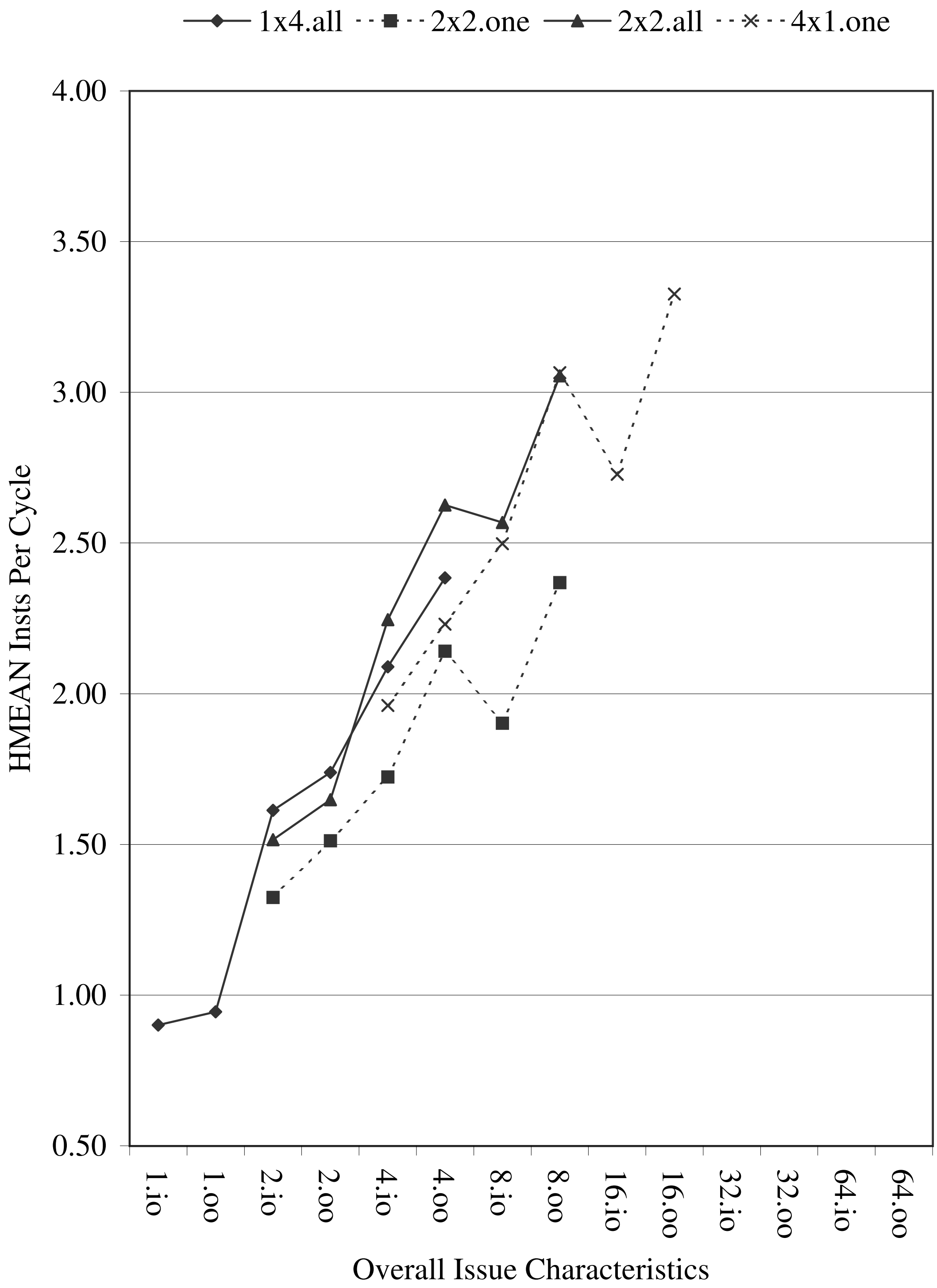


Figure 4.18: Harmonic mean of IPC using 4 tasks for SPEC CFP95 benchmarks.

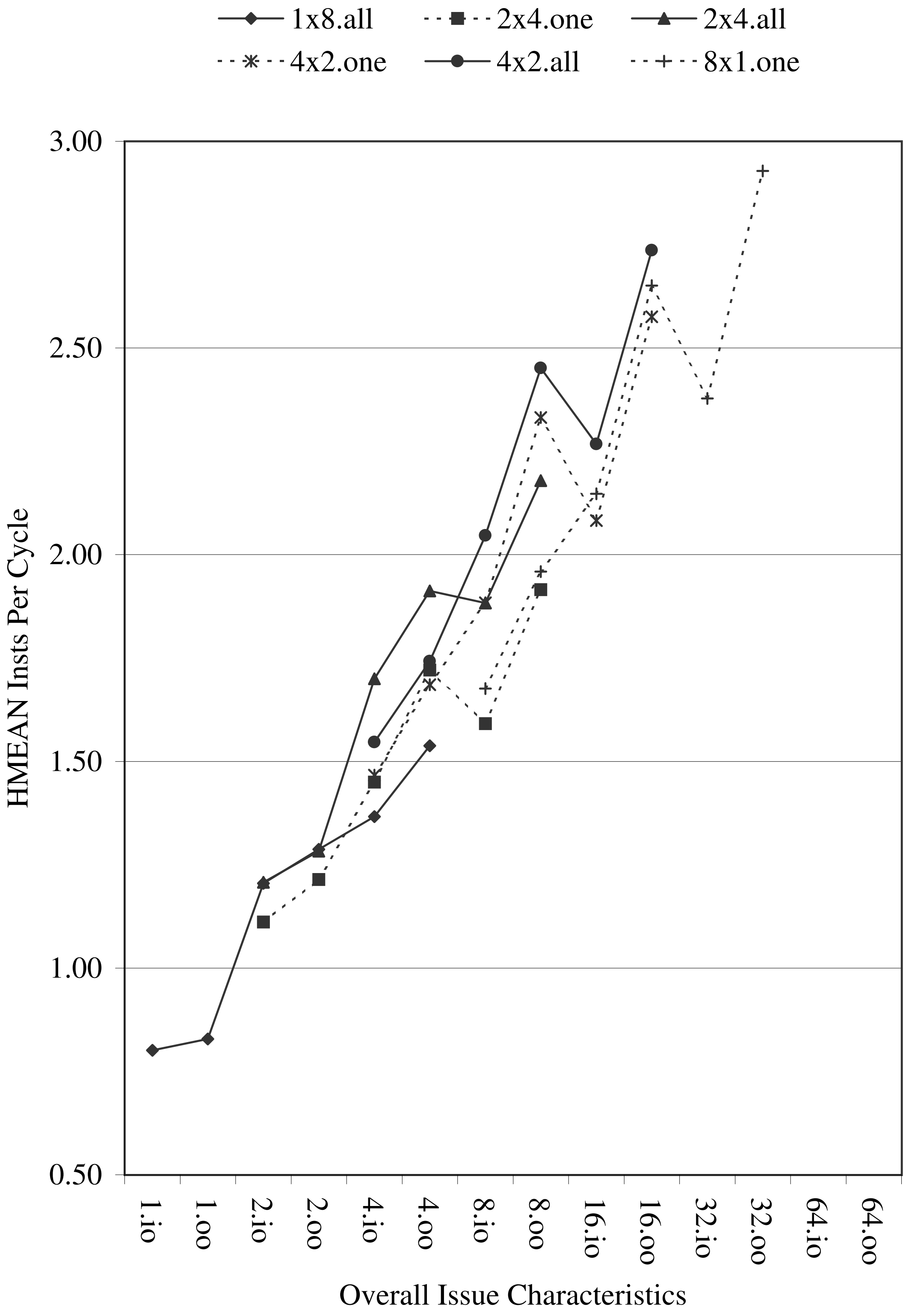


Figure 4.19: Harmonic mean of IPC using 8 tasks for SPEC CINT95 benchmarks.

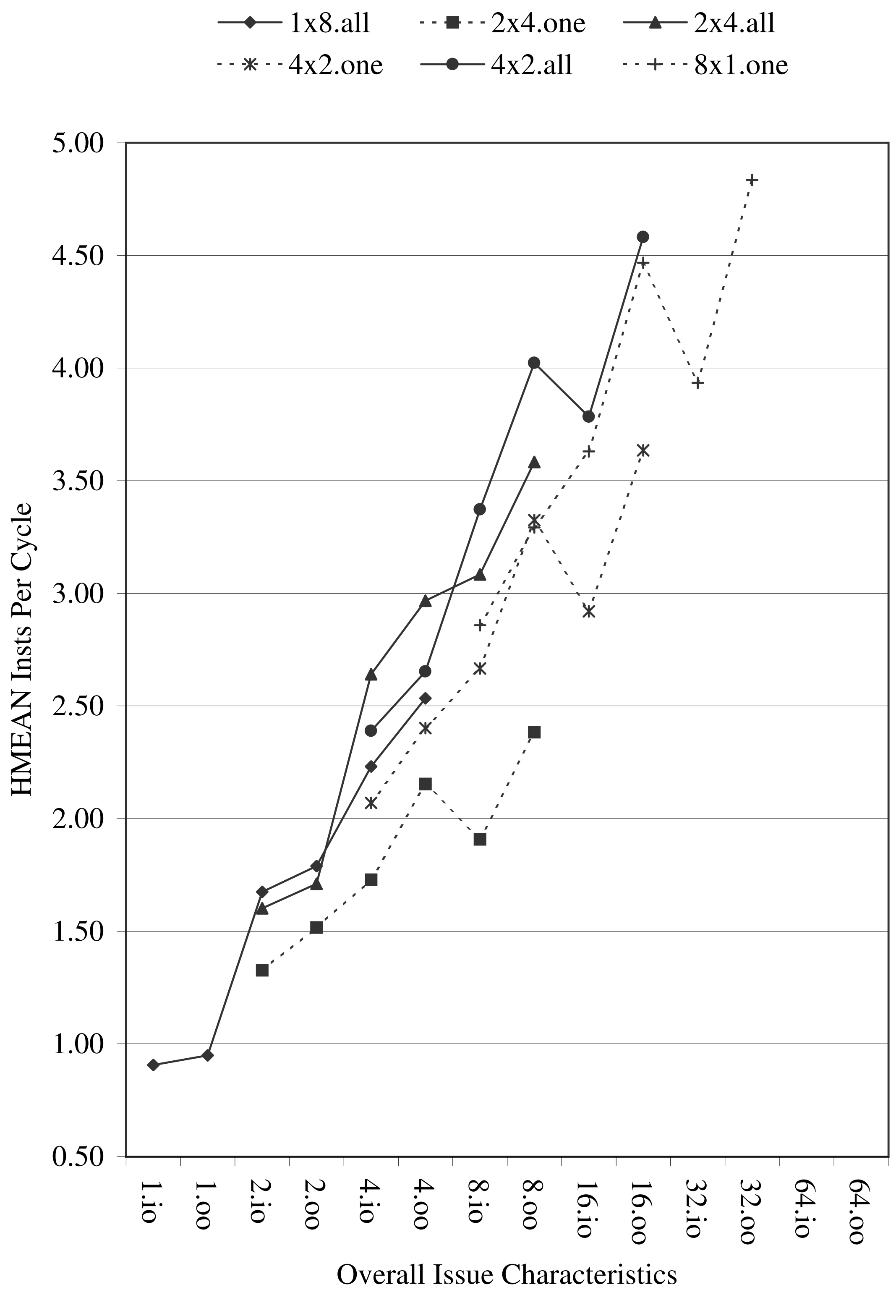


Figure 4.20: Harmonic mean of IPC using 8 tasks for SPEC CFP95 benchmarks.

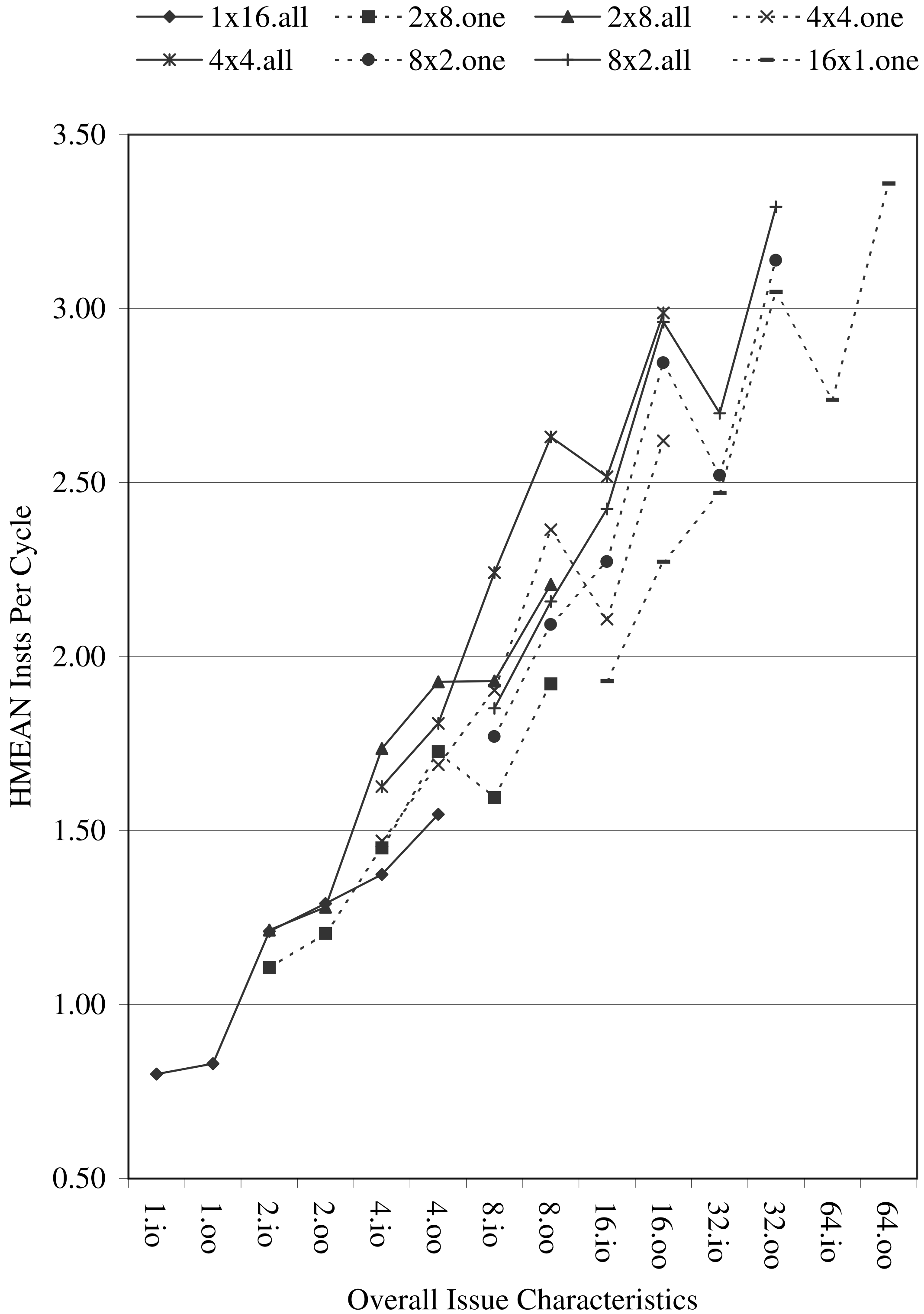


Figure 4.21: Harmonic mean of IPC using 16 tasks for SPEC CINT95 benchmarks.

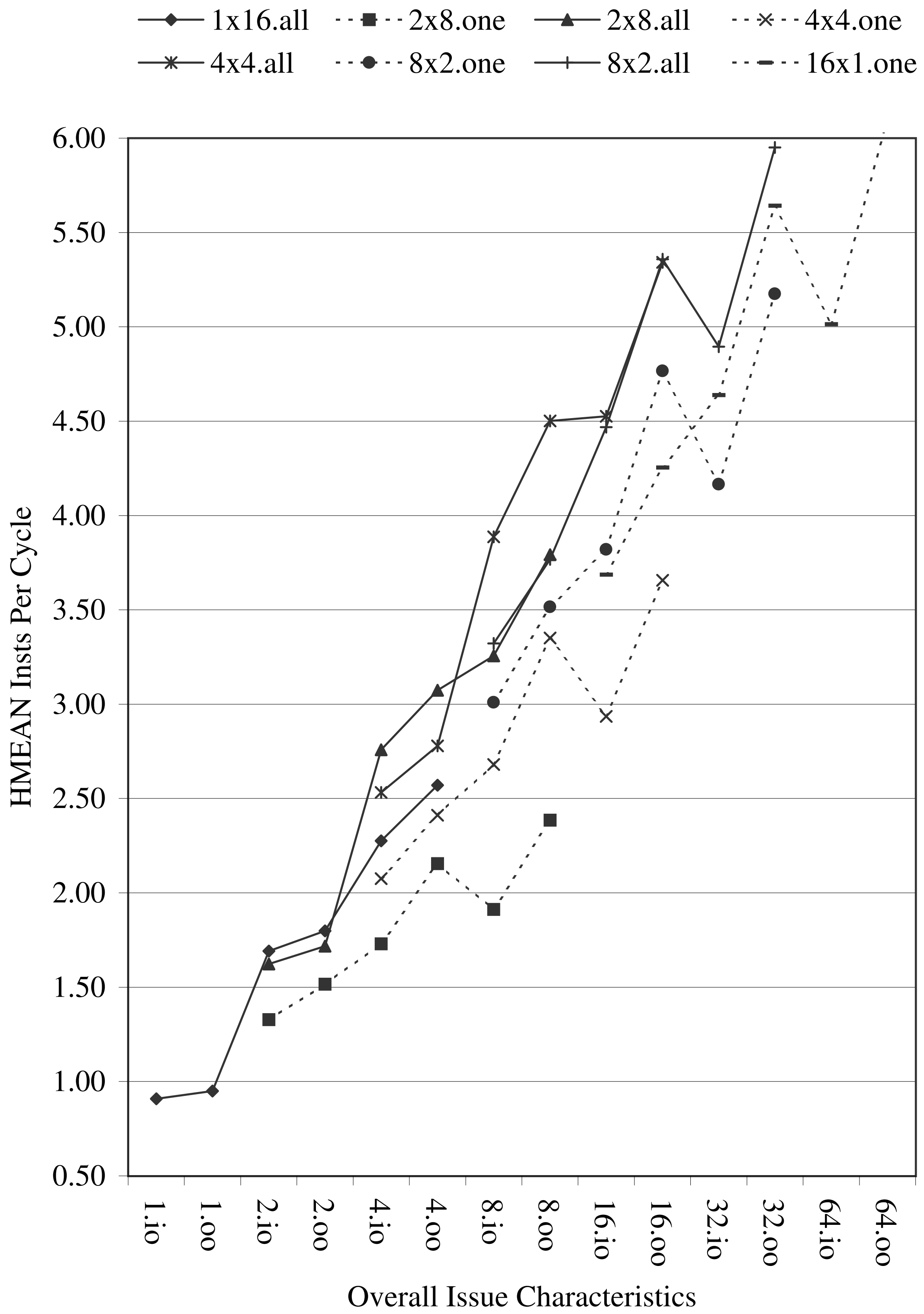


Figure 4.22: Harmonic mean of IPC using 16 tasks for SPEC CFP95 benchmarks.

4.6.4.2 Different Number of Tasks

It is somewhat involved to consider processor configurations that support a different number of tasks. One way to do so is to consider performance ranges and identify what configurations achieve similar performance. Even taking this approach, there are still many different configurations with similar performance. Yet, if the focus is on a specific range, then the number of configurations to consider is small enough that some of the more noteworthy ones may be contrasted. The configurations identified are only meant to point out that there are many ways to achieve comparable performance in a multiscalar processor, not to suggest which ways are the most cost-effective. Such a determination is beyond the scope of this work.

Considering the integer benchmarks, the range of performance below 1.50 instructions per cycle is too low of an overall issue to be of interest for this discussion. Likewise, the range of performance above 2.50 instructions per cycle is only possible with many tasks and/or high issue. So, for the purpose of this discussion, the range of 1.50 to 2.50 instructions per cycle is a suitable one to consider. Overall, for the integer benchmarks, configurations with 4 processing units running tasks one-at-a-time for one-to-one mappings (no other option exists) and running all-at-once for many-to-one mappings achieve the best performance over the range of issue characteristics that may be supported by them.

At the low end of the range, around 1.50 instructions per cycle, 1 4-wide out-of-order processing unit running 8 tasks all-at-once provides comparable performance to 4 1-wide out-of-order processing units running 4 tasks one-at-a-time. At the middle of the range, around 2.00 instructions per cycle, 4 1-wide in-order processing units running 4 tasks one-at-a-time provides comparable performance to 16 1-wide in-order processing units running 16 tasks one-at-a-time. At the high end of the range, around 2.50 instructions per cycle, 4 2-wide out-of-order processing units running 8 tasks all-at-once provides comparable performance to 8 2-wide in-order processing running 16 tasks all-at-a-once.

Considering the floating point benchmarks, the range of performance below 3.00 instructions per cycle is too low of an overall issue to be of interest for this discussion. Likewise, the range of performance above 4.50 instructions per cycle is only possible with many tasks and/or high issue. So, for the purpose of this discussion, the range of 3.00 to 4.50 instructions per cycle is a suitable one to consider. Overall, for the floating point benchmarks, configurations with 4 processing units running tasks one-at-a-time for one-to-one mappings and running all-at-once for many-to-one mappings nearly always achieve the best performance over the range of issue characteristics that may be supported by them.

At the low end of the range, around 3.00 instructions per cycle, 2 4-wide out-of-order processing units running 4-tasks all-at-once provides comparable performance to 8 1-wide in-order processing units running 16 tasks one-at-a-time. At the middle of the range, around 3.75 instructions per cycle, 4 4-wide in-order processing units running 16 tasks all-at-once provides comparable performance to 8 4-wide in-order processing units running 8 tasks one-at-a-time. At the high end of the range, around 4.50 instructions per cycle, 4 4-wide out-of-order processing units running 8 tasks all-at-once provides comparable performance to 16 2-wide in-order processing units running 16 tasks one-at-a-time.

4.7 Summary

This chapter studied instruction and data processing of a multiscalar processor. The basics of how a multiscalar processor exposes and exploit instruction-level parallelism using multiple tasks and processing units was explained. To this explanation was added a discussion of the processing phases performed during the lifetime of a task. Using these concepts, the outlook for performance gains and losses was established and the techniques that might be needed to improve the interaction of tasks and processing units were identified. In particular, a multiscalar processor may suffer from performance losses due to load balance, limited exposed parallelism, and/or resource under-utilization. All of these factors were then considered in term of the processing unit design. A comprehensive description of processing unit design was given with the details divided among inter-task and intra-task aspects.

4.7.1 Inter-Task Aspect

For the inter-task aspect, the issues of mapping policy, which dictates how tasks are assigned to processing units, and running policy, which dictates how many tasks a processing unit may run concurrently, were identified as key design choices. In terms of mapping policy, two policies – round-robin and back-to-back – were described, but only the former was evaluated. In terms of the running policy, two policies – one-at-a-time and all-at-once – were described and evaluated.

In terms of the inter-task aspect, it was found that linear performance improvements may be achieved as the number of tasks in concurrent execution were doubled, assuming that tasks were assigned one-to-one to processing units. For SPEC CINT95 programs, the harmonic mean of the instructions per cycle ranged roughly from 1.5 to 3.5, using from 2 to 16 tasks using and, as a result of the one-to-one assignment, an equal number of processing units. For SPEC CFP95 programs, the harmonic mean of the instructions per cycle ranged roughly from 2.0 to 6.0, using from 2 to 16 tasks and likewise an equal number of processing units.

Though the performance results were consistent with performance expectations, the modest performance improvement as well as the steady decline in resource utilization relative to the cost, called for the consideration of alternative processing unit designs. The root cause of the load balancing difficulties was deemed to be in the processing unit design rather than in the multiscalar execution model. As a result, a new processing unit design that supported many-to-one task to processing unit assignment was examined as an alternative with more desirable cost-performance properties.

Such a design may lever techniques similar to traditional multithreading to allow each processing unit to execute multiple tasks at the same time. Though the design was based on traditional multithreading, the handling of control and data dependences had to be adapted to support the multiscalar execution model since tasks, unlike threads, cannot be assumed to be control or data independent. A variety of different organizations that supported up to a total of 16 tasks and 16 processing units, but offered different cost-performance points were considered.

After studying the different multithreading organizations, it was determined that a configuration of 4 processing units running 4 tasks all-at-once per processing unit (a total of 16 tasks) was able to nearly match (90%) the performance of the configuration with the highest performance, 16 processing units running 1 task per processing unit, but at a fraction of the cost (25%). In particular, for the SPEC CINT95 programs, the harmonic mean of the instructions per cycle reached roughly 3.0 (as compared to 3.5); for the SPEC CFP95 programs, the harmonic mean of the instructions per cycle reached roughly 5.5 (as compared to 6.0).

4.7.2 Intra-Task Aspect

For the intra-task aspect, the instruction window, the instruction selection and issue, as well as the instruction execution characteristics of the individual processing units were identified as key design choices. In terms of the intra-task aspect, the performance improvements that may be realized depend on the particular microarchitectural characteristics of the processing units. It is easiest to describe how performance was impacted by focusing on each characteristic (among instruction window, instruction selection and issue, and instruction execution) in turn. For the instruction window, the effects of different in-flight and on-deck configurations were described and evaluated. For the instruction selection and issue, the impact of multiple instruction issue and dynamic instruction scheduling were described and evaluated. Finally, for the instruction execution, the effects of the number and type of functional units on execution was described and evaluated.

First, the instruction window of each processing unit does affect the performance, but not to as great an extent as in conventional superscalar processors, since a multiscalar processor extracts instruction-level parallelism via the collective windows of all of the processing units rather than via the individual window of one processing unit (as a superscalar processor does). As a result, a multiscalar processor using processing units each with an instruction window of only 16 instructions in-flight (decoded) and 8 instructions on-deck (decoded, but not issued), achieved comparable performance to an instruction window of 64 instructions in-flight and 64 instructions on-deck. In particular, for SPEC CINT95 programs, the performance was the same, and for the SPEC CFP95 programs, the performance was only about 10% less comparing these small and large window designs.

Second, the instruction selection and issue for each of these windows was varied among in-order and out-of-order characteristics for issue widths of 1, 2, and 4 instructions per cycle. It was found that performance grows steadily, for both SPEC CINT95 and SPEC CFP95 programs for both in-order and out-of-order designs. Furthermore, the growth rate going from 1-wide to 2-wide is more significant than going from 2-wide to 4-wide, for both the in-order and out-of-order designs. These performance characteristics were observed among multiscalar processor configurations of 1, 2, 4, 8, and 16 processing units, with the effects on overall performance more significant for more processing units. Moreover, in comparing the in-order with the out-of-order designs it was observed that the out-of-order growth rate is higher than the in-order growth rate. As a result, there is a crossover point (at 2-wide out-of-order and 4-wide in-order) in terms of performance, where a lower issue width out-of-order processing unit performs better than a higher issue width in-order processing unit.

Third, the instruction execution capability in terms of numbers of different types of functional units per processing unit has a direct impact on performance, for both SPEC CINT95 and SPEC CFP95 programs. For this characteristic, the functional units of a processing unit were classified as handling compute (integer and floating point), memory, and control instructions. The number of functional units of each type was varied for issue widths of 1, 2, and 4 instructions per cycle and compared with designs using uniform (able to execute any type) functional units. For both the SPEC CINT95 and SPEC CFP95 programs, it was observed that for multiple instruction per cycle issue widths, the availability of at least 2 computation and 2 memory functional units was necessary to provide performance comparable to that of uniform functional units, for the 2-wide and 4-wide issue widths; the use of only 1 control functional unit was sufficient in all cases. These observations indicated that a processing unit must be provided with at least 2 memory and computation functional units for effective execution.

Chapter 5

Instruction Supply

The instruction and data processing in a multiscalar processor must be supported by appropriate instruction and data supplies, since effective extraction of instruction-level parallelism requires a plentiful amount of both. The next chapter will study the data supply portion. This chapter studies the instruction supply of a multiscalar processor, with the first half concentrating on background and the second half on design and evaluation.

Section 5.1 gives an overview of the components, hierarchical predictor and instruction memory, used to supply tasks and instructions for instruction and data processing. Section 5.2 explains program sequencing in multiscalar processors. Section 5.3 discusses the instruction supply requirements of multiscalar processors. Section 5.4 reflects on concerns/challenges in designing the instruction supply components.

Section 5.5 investigates hierarchical predictor design for the instruction supply. Section 5.6 provides an evaluation of the hierarchical predictor designs. Section 5.7 investigates instruction memory design for the instruction supply. Section 5.8 provides an evaluation of the instruction memory designs. Section 5.9 gives a summary of this chapter.

5.1 Overview

The instruction supply in a multiscalar processor is provided by a collection of predictors and caches to support the processing units, as shown in Figure 5.1. Each of the individual components is similar to its counterpart in conventional processors. However, the organization is somewhat different to provide support for the multiscalar concept of dividing the dynamic instruction sequence into tasks and executing these tasks on multiple processing units in parallel.

5.1.1 Basics

The instruction supply relies on control flow prediction and instruction memory access, like a conventional processor. Yet, each of these mechanisms involves an inter-task aspect and an intra-task aspect, unlike a conventional processor. The components of the control flow prediction and the instruction memory may be classified according to what role, either inter-task or intra-task, that each plays in supplying instructions to the processing units.

The inter-task portion of the instruction supply exposes the sequence of tasks that constitute the window on the dynamic instruction sequence and establishes the order among tasks in concurrent execution. The intra-task portion of the instruction supply exposes the dynamic

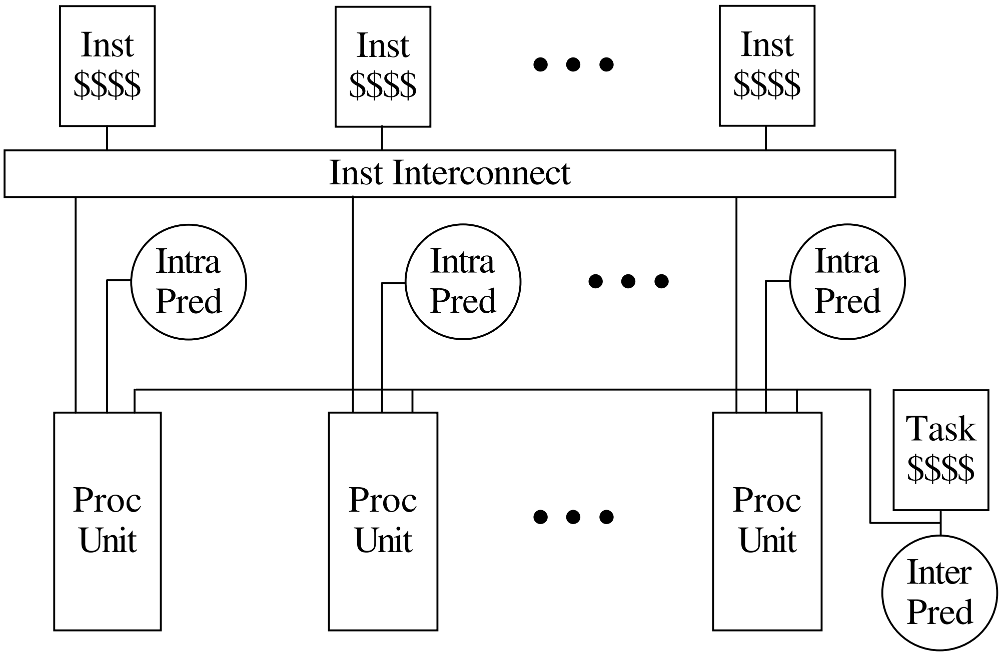


Figure 5.1: The organization of the instruction supply components and the processing units in a multiscalar processor.

instruction sequence of each task individually, and among all tasks collectively the dynamic instruction sequence of the program.

5.1.1.1 Inter-Task Components

The inter-task components of the instruction supply consist of an inter-task predictor (the control flow prediction component, **Inter Pred**, in the figure) and a task cache (the instruction memory component, **Task $$$$**, in the figure). The inter-task predictor and the task cache are accessed at the same time with the address of the current task. The inter-task predictor speculates the task exit of the current task from among those identified by the compiler. The task cache is accessed to obtain the task descriptor (provided by the compiler), for the current task.

Using static control information in the task descriptor and perhaps dynamic control information maintained by the processor, the address which corresponds to the entry point for the next task is determined. (Furthermore, using static register data information in the task descriptor, the register dependence linkages are set up for the next task, as discussed in the next chapter; the memory dependence linkages are handled using dynamic memory data information, as discussed in the next chapter.) This process repeats each cycle, moving through the program in this manner task-by-task.

5.1.1.2 Intra-Task Components

The intra-task components of the instruction supply consist of intra-task predictors (the control flow prediction components, **Intra Pred**, in the figure) and instruction caches (the instruction memory components, **Inst $$$$**, in the figure). The intra-task predictors are comparable to those used in conventional processors. Likewise, the instruction caches are similar to those used in conventional processors, but their organization must provide enough bandwidth to support all of the processing units. The connectivity between the processing units and the instruction caches provided by the instruction interconnect (**Inst Interconnect** in the figure) is the main determinant of how the bandwidth is provided and with what latency.

Using its own intra task predictor, each processing unit speculates the path it follows as it executes a task and provides the instruction addresses for this path. Then, using these instruction addresses, each processing unit accesses the instruction caches, which are maintained as either a private or shared organization depending on the interconnect, to obtain the actual instructions that make up its task(s). This process repeats each cycle, with processing units moving through tasks in this manner instruction-by-instruction.

5.1.2 Insights

The combined behavior of the control flow prediction and instruction memory mechanisms determines the characteristics of program sequencing and the performance that may be obtained in terms of the instruction supply. Overall, the hierarchical instruction supply of a multiscalar processor using multiple program counters alleviates the bottlenecks associated with the sequential instruction supply of conventional processors using a single program counter. Though there is a logical distinction between the inter-task components and the intra task components, it is more convenient for the purposes of this study to make a logical distinction instead between the hierarchical predictor and the instruction memory, and then make the distinction between the inter-task aspect and the intra-task aspect within the context of each mechanism.

5.1.2.1 Hierarchical Predictor

The design and evaluation of the hierarchical predictor performed in this chapter demonstrates that both the inter-task and intra-task aspects can be provided using components based on existing prediction technology. In this study, the design of the hierarchical predictor is considered in terms of using several well-known (address-based, path-based, global-pattern-based, and self-pattern-based) as well as perfect prediction schemes for both inter-task and intra-task predictors. First, the inter-task and intra-task predictors are evaluated in isolation of one another using each of these predictor designs to determine the best scheme for each. Second, the best inter-task scheme and the best intra-task scheme are used to examine the effect of the hierarchical predictor on overall performance when different amounts of prediction state devoted to each predictor.

Comparing the inter-task and intra-task predictors generally, both inter-task and intra-task prediction can be important performance factors. Nonetheless, the results of this study indicate (in reference to the SPEC CINT95 programs where there is appreciable performance differences among both inter-task and intra-task predictors, rather than the SPEC CFP95 programs where there is none) that the overall performance is more sensitive to the inter-task than to the intra-task prediction accuracy. In addition, comparing the predictors studied with perfect predictors, there is considerable performance improvement possible (as much as 35%) for the SPEC CINT95 benchmarks, but almost none for the SPEC CFP95 benchmarks. Moreover, the potential improvement increases with the number of concurrent tasks supported by a multiscalar processor and is more significant for inter-task than for intra-task predictors.

Comparing the different prediction schemes studied for inter-task and intra-task predictors specifically – path-based, global-pattern-based, self-pattern-based, and addressed-based – the path-based and the global-pattern-based schemes consistently performed best across the range of SPEC CINT95 and SPEC CFP95 programs for both inter-task and intra-task prediction. On the basis of these results as well as other considerations of design complexity, this study identifies the combination of a path-based inter-task predictor with a global-pattern-based intra-task predictor as the best hierarchical predictor design for the multiscalar processors considered in this thesis. Furthermore, with respect to this hierarchical predictor design, varying the amount of prediction state from 8K-entry to 64K-entry for each individual predictor (inter-task and intra-task) shows no appreciable difference in prediction accuracy among these sizes.

5.1.2.2 Instruction Memory

The design and evaluation of the instruction memory performed in this chapter demonstrates that both the inter-task and intra-task aspects of the instruction memory can be provided using components based on existing cache technology – a task cache for inter-task and an instruction cache for intra-task. Thus, for the experiments in this study, both the task cache and instruction cache are evaluated in terms of miss rate, effective access latency, and overall impact on performance for a variety of sizes and associativities. Moreover, the instruction cache is evaluated using two different high-bandwidth organizations, interleaved shared cache and duplicated private cache, as well as a hybrid organization using both private and shared caches. To complement these experiments, this study also considers ideal versions of the task and instruction caches in order to gauge the magnitude of performance improvement possible via instruction memory access.

With respect to the miss rate oriented cache experiments, both task and instruction caches exhibit typical cache-like behavior. That is, the miss rate of either cache decreases steadily as the cache size increases, leveling off once the working set fits into the cache. Likewise, the use of associativity (with diminishing returns as with the cache size) improves the miss rate of either cache, with most of the improvement moving from 1-way associativity to 2-way associativity and only somewhat of an improvement moving from 2-way to 4-way associativity. However, it is significant to note that private and shared instruction cache organizations only provide comparable performance when the individual storage of each private cache is comparable to the overall storage of the shared instruction cache. The hybrid instruction cache is able to mitigate this situation somewhat, with more success for less processing units (with the break-even point around 4 processing units).

With respect to the latency oriented cache experiments, both task and instruction caches may play a significant role in terms of degrading the overall processor performance. That is, as the effective access latency increases, either due to an increase in access time to service misses or due to an increase in access time to service hits, overall performance decreases. In particular, the performance for the SPEC CINT95 benchmarks drops off far more significantly than for the SPEC CFP95 benchmarks. Moreover, this behavior is far more pronounced in the task caches than it is in the instruction caches. In fact, instruction caches with relatively low miss rates, are capable of tolerating relatively high effective access latencies, while task caches are not, regardless of their miss rates.

Overall, the results of this study indicate that both task and instruction caches can be important performance factors. Nevertheless, comparing the two, the results suggest that performance is more sensitive to the task cache behavior than to the instruction cache behavior. Such behavior may be explained by the fact that task cache accesses expand the overall instruction window in a serial, dependent manner, while instruction cache accesses expand the individual instruction windows of only their corresponding processing unit in a parallel independent, manner. Moreover, the substitution of ideal task and instruction caches for real ones offers no discernible performance improvement so long as the real task and instruction caches are not extremely small in size. Thus, according to the results of this study, there is no clear need for radical changes to conventional cache designs in order to provide effective instruction memory access for a multiscalar processor.

5.2 Program Sequencing

With the overview of the instruction supply just presented in mind, it is worth switching gears to consider what is the motivation for using this alternative hierarchical approach to the instruction supply instead of the traditional sequential approach.

The cycle of instruction address generation and instruction memory access used to supply instructions (alluded to above in terms of the inter-task and intra-task components of a multiscalar processor) is often referred to as program sequencing. In a conventional processor (such as a superscalar processor), program sequencing is usually performed with a single program counter.

This program counter tracks the point to which the instruction supply mechanism has progressed in the dynamic instruction sequence. The program counter is advanced through the static representation of the program according to the dynamic path of execution. As the program counter is advanced, the instruction memory is accessed to obtain the instructions (and perhaps other information) that constitute the dynamic instruction sequence for the path of execution.

In processor implementations that exploit instruction-level parallelism dynamically, program sequencing is usually driven by aggressive control flow prediction and expeditious instruction memory access. The goal in this case is to build up a window on the dynamic instruction sequence from which to find instructions for concurrent execution. The control flow prediction speculates the dynamic path of program execution. The instruction memory access delivers the instructions (and perhaps other information) on this speculated dynamic path to the processor.

The components that perform control flow prediction and instruction memory access must usually be designed for high-bandwidth and low-latency operation so that many instructions may be supplied quickly to the processor. These characteristics are often needed to allow a dynamically scheduled processor to expose a window on the dynamic instruction sequence of sufficient magnitude to exploit significant amounts of instruction-level parallelism.

5.2.1 Single versus Multiple Program Counters

The actual form that the demand for high-bandwidth and low-latency in the instruction supply takes depends on the operation of the processor.

A conventional processor, such as a superscalar processor, uses a single program counter to perform program sequencing because it views the dynamic instruction sequence as a single region. As such, a conventional processor follows a single flow of control and exposes a single window on the dynamic instruction sequence.

In contrast, a multiscalar processor divides the dynamic instruction sequence, and therefore views it as multiple regions (specifically, as tasks in this work). It uses multiple program counters, one per task, to perform program sequencing in these regions. As such, a multiscalar processor follows multiple flows of control and exposes multiple windows on the dynamic instruction sequence.

Though instruction bandwidth is usually increased for both approaches by predicting multiple control flow points per cycle and accessing multiple instruction cache blocks per cycle, the design of the components to do so may be quite different according to whether a single program counter or multiple program counters are used.

5.2.1.1 Using a Single Program Counter

Using a single program counter, as in a superscalar processor, the goal of the instruction supply is to deliver a single contiguous stream of instructions to the processor each cycle. In order to increase the bandwidth with which instructions are supplied, it is necessary to increase the length of the stream. The length of the instruction stream may be increased by predicting several consecutive control flow points and performing several instruction memory accesses each cycle. However, the mechanisms needed to do so using a single program counter may be involved due to the serial nature of performing these operations, from both control flow prediction and instruction memory access standpoints.

A typical control flow predictor is designed to predict only one control flow point per cycle. Some attention has been given to predictor designs that perform two or more successive predictions per cycle [19, 103], but these schemes seem to be limited to at most two or three predictions per cycle. Moreover, these schemes often provide multiple predictions at the cost of lower accuracy and/or higher complexity. Ultimately, the control flow predictor must support more predictions per cycle than these schemes do. However, a straightforward extension of these schemes to perform an arbitrary number of predictions per cycle appears to be difficult, implying entirely new prediction schemes may be required for single program counter sequencing. In addition, it is likely that such complex predictors may entail long prediction latency as compared to simpler alternatives. Yet, even if multiple predictions per cycle can be made in an effective and efficient manner, there are still complications with regard to the instruction memory accesses driven by them.

Though it is relatively straightforward to support multiple instruction memory accesses per cycle with an appropriate cache organization, there are complications with forming a single contiguous stream from the instructions delivered by multiple arbitrary cache accesses. The layout of instructions in the static program does not usually match the dynamic path of execution. Therefore, in order to turn multiple cache accesses into a contiguous stream of instructions, the instructions returned from the caches must be merged and arranged properly. In general, this operation requires an extensive alignment and reduction network that may be both complex and slow. There has been some work on this problem [16, 102], but as with multiple predictions, it seems to be limited to at most two are three accesses per cycle, with latency increasing according to the number of accesses handled. Trace caches have been proposed to keep the latency of merging and arranging off of the critical path by caching contiguous instruction traces [76], but their utility remains an open question because the practical restrictions on trace length may limit the bandwidth and the actual hit rate of the trace cache may limit its effectiveness.

5.2.1.2 Using Multiple Program Counters

Using multiple program counters, as in a multiscalar processor, the goal of the instruction supply is to deliver multiple contiguous streams of instructions to the processor (for each of the processing units) each cycle. In order to increase the bandwidth with which instructions are supplied, it is necessary to increase the length of the streams, to increase the number of streams, or both. Due to the complexities already described with increasing the length of a single stream, it is preferable to increase the number of streams, which for a multiscalar processor implies an increase in the number of tasks and/or processing units. This strategy provides a balanced approach in which increases in instruction and data processing are matched with increases in instruction supply. Moreover, the mechanisms needed to do so are straightforward due to the parallel nature of performing control flow prediction and instruction memory access in an independent manner among the multiple program counters.

Of course, this type of parallel program sequencing (at the intra-task level) does require some serial program sequencing (at the inter-task level) to provide the initial values (task entry points) for the multiple program counters. However, the mechanisms involved for the serial part are comparable to those used in conventional processors and need only perform one control flow prediction and instruction memory access per cycle. Moreover, for the parallel part, the complex mechanisms developed to support multiple consecutive operations for control flow prediction and instruction memory access may be replaced with simple mechanisms to support a single control flow prediction and instruction memory access per cycle. To be specific, these simple mechanisms may be replicated, one for each program counter, to perform multiple control flow predictions and instruction memory accesses in a parallel rather than a serial manner. In this way, the potential complexity and latency difficulties of increasing instruction bandwidth that plague single program counter sequencing may be held in check.

5.3 Multiscalar Requirements

In order to clarify the abstract concept of using multiple program counters, it is helpful to describe in concrete terms the multiscalar requirements for the instruction supply.

5.3.1 Control Flow Prediction

As illustrated in Figure 5.2, inter-task (**Inter Pred** in the figure) predictions are performed in a pipelined fashion one after another. Thus, the inter-task components of the hierarchical predictor need only support one control flow prediction per cycle. Since each inter-task prediction provides another program counter (task entry) for sequencing, it triggers a string of intra-task (**Intra Pred** in the figure) predictions that are performed one after another for a particular task, but in parallel with the predictions for other tasks. Thus, the intra-task components of the hierarchical predictor must support multiple control flow predictions per cycle, one for each task. However, as described earlier, each sequence of predictions is independent and may be performed by individual intra-task predictors that need only support one control flow prediction per cycle.

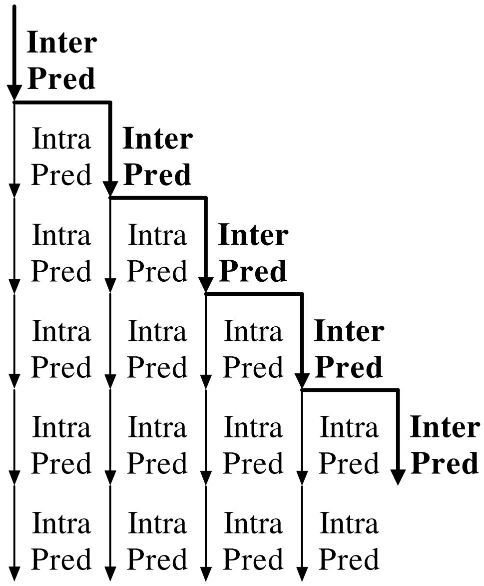


Figure 5.2: The control flow prediction requirements to support a multiscalar processor.

5.3.2 Instruction Memory Access

As illustrated in Figure 5.3, task cache (**Task $$$$** in the figure) accesses are performed in a pipelined fashion one after another, corresponding to the inter-task predictions. Thus, the inter-task components of the instruction memory need only support one instruction memory access per cycle. Since each task cache access provides another task for sequencing, it triggers a string of instruction cache (**Inst $$$$** in the figure) accesses, corresponding to the intra-task predictions, that are performed one after another for a particular task, but in parallel with the accesses for other tasks. Thus, the intra-task components of the instruction memory must support multiple instruction memory accesses per cycle, one for each task (or processing unit if multiple tasks are assigned). However, as described above, each sequence of instruction memory accesses is an independent stream and may be serviced by an appropriate cache organization without the need to merge and arrange the instructions returned from the multiple accesses into a single contiguous stream, as is needed for conventional processors.

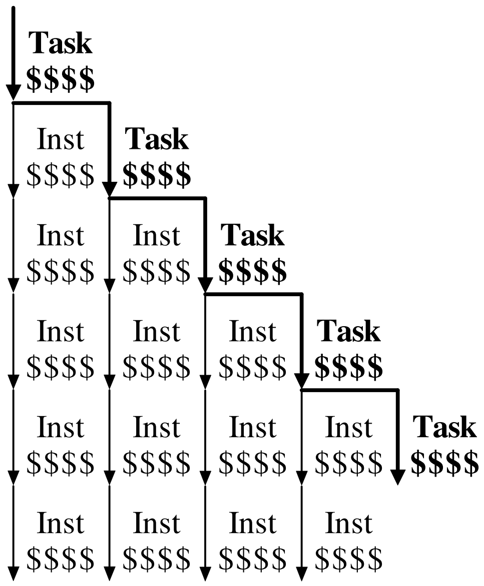


Figure 5.3: The instruction memory access requirements to support a multiscalar processor.

5.4 Concerns/Challenges

The combined behavior of the control flow prediction and instruction memory access mechanisms determine the characteristics of program sequencing and the performance that may be obtained in terms of the instruction supply. Overall, the hierarchical instruction supply of a multiscalar processor using multiple program counters alleviates the bottlenecks associated with the sequential instruction supply of conventional processors using a single program counter. Nevertheless, there are still some concerns/challenges associated with an instruction supply of this design that must be addressed to ensure it functions effectively and efficiently.

5.4.1 Hierarchical Predictor

A side-effect of dividing the dynamic instruction sequence into tasks is that the control flow points of the program are divided as well. Consequently, the control flow points seen by the multiple predictors of a multiscalar processor do not correspond in number and/or sequence to the control flow points seen by the single predictor in a conventional processor. Moreover, the characteristics of the control flow points that are seen are different depending on whether the inter-task predictor or the intra-task predictors are involved.

At the inter-task level, predictions are not made for individual control flow points. Tasks, in general, are bounded by multiple control flow edges (otherwise tasks would be little more than basic blocks). Thus, the inter-task predictor selects from among multiple (more than two) control flow targets. At the intra-task level, because control flow is divided amongst many processing units, the intra-task predictors observe only discontinuous portions of the control flow behavior of a program. These differences raise concern about the accuracy with prediction may be performed.

In the case of static prediction schemes, these factors ought to have little effect. However, in the case of dynamic prediction schemes, which rely on the record of control flow behavior, prediction accuracy might suffer. The observable control flow from the inter-task viewpoint is the outcome of selected (not usually adjacent) control flow points. The observable control flow from the intra-task viewpoint is the remaining control flow points between those selected as task boundaries. Thus, the inter-task and intra-task prediction decisions may have to be made from incomplete information; only the control flow record of tasks which executed on the particular processing unit can be easily maintained.

Considering the role speculation plays in multiscalar execution, the accuracy is an important factor for overall performance. The inter-task prediction mechanism must be very accurate to give good performance. Poor inter-task control flow prediction limits the extent of the dynamic window of instructions, decreasing the exposed parallelism across tasks. The intra-task prediction mechanism can tolerate inaccuracy, but accuracy is still an important factor for good performance. Poor intra-task control flow prediction delays the execution of individual tasks, and thereby aggravates control and data dependences between tasks, increasing the execution time overall.

5.4.2 Instruction Memory

With respect to the instruction memory, the concern is simply the bandwidth and latency of access. Obviously, the use of cache memory is the best known method to increase bandwidth and/or to decrease latency. So long as spatial and/or temporal locality exists in the dynamic instruction sequence, caches represent an effective technique for this purpose. However, in a multiscalar processor, there are two types of access, inter-task and intra-task. Because each type of access has different characteristics from one another as well as from the type of accesses performed in conventional processors, there is some concern about the design of the caches to service them.

5.4.2.1 Task Cache

The task cache that services the inter-task accesses is relatively straightforward because it need only support a single access per cycle. Nonetheless, the task cache is the critical path in terms of exposing the sequence of tasks that constitute the dynamic instruction sequence. The bandwidth to the cache is not a concern, but the latency to the cache certainly might be because task assignment can only continue at the rate task descriptors can be made available from the task cache. An increase in the task cache access latency can have a multiplicative effect on the number of processing unit cycles spent as idle time when the processor must wait to assign tasks to execute.

The impact of these idle processing units cycles on overall performance depends in part on the tasks themselves and in part on their execution. If the tasks are big relative to the idle cost, then the idle cost may be only an insignificant fraction of the overall execution time. However, if the tasks are small relative to the idle cost, then the processing unit idle time may dominate the execution time. Moreover, the frequency of events such as dependence violations can influence the impact of the idle processing unit cycles. To be specific, such events cause the sequence of task to be flushed from the processor. Therefore, regardless of whether tasks are big or small, the processor must incur the latency of task re-assignment to the processing units.

5.4.2.2 Instruction Cache

The instruction caches that service the intra-task accesses are somewhat more involved because of the need to support multiple accesses per cycle. In particular, the concern is how to make the trade-off between hit rate and hit latency in providing adequate bandwidth to the processing units using (two widely studied high-bandwidth cache organizations) either a shared instruction cache scheme (of interleaved caches) or a private instruction cache scheme (of duplicated caches). The nature of multiscalar execution is such that not all instructions from the dynamic instruction sequence factor into the instruction memory access characteristics of an individual processing unit. Each processing unit only executes the portion of the dynamic instruction sequence represented by its task.

As a result, the instruction memory access characteristics of private caches, one connected to each processing unit, may be quite different from those of shared caches connected to all processing units. To be specific, accesses of the same instructions may miss in different private caches and/or replacements of instructions may occur before a profitable reuse. In contrast, though the accesses may hit in the shared caches, there is an inherent latency of traversing the interconnect in such an organization as well as the additional latency due to handling conflicts with multiple accesses to the same cache. However, instruction memory accesses may occur in parallel among processing units and may be pipelined within a processing unit, so the overall performance may not be as sensitive to instruction cache latency as to task cache latency.

5.5 Hierarchical Predictor Design

Up to this point, the control flow in a program has not been differentiated in any particular way. However, there are usually many different types of control flow in a program: conditional branches, unconditional branches, indirect jumps, procedure calls, procedure returns, etc. As different types of control flow exhibit different types of behavior, no single mechanism suffices for all cases. A comprehensive prediction mechanism relies on a collection of techniques to provide the best performance: (i) branch prediction for conditional branches [48, 54, 66, 80, 103], (ii) call/return stack for procedure calls and procedure returns [42], and (iii) branch target buffer for indirect changes in control [12, 48].

The focus in this work is on predictors as unified mechanisms to generate addresses for instruction memory accesses. As a result, these predictor components are considered collectively as parts of a comprehensive control flow prediction mechanism rather than individually as separate mechanisms (though all of the different types of control flow are handled as appropriate). Recall, that a multiscalar processor employs a hierarchical prediction mechanism, with a single inter-task predictor for all processing units that drives multiple intra-task predictors for each processing unit.

The inter-task aspect of hierarchical prediction was originally reported in the literature by Pnevmatikatos, Franklin, and Sohi [70], and later by Jacobson, Bennett, Sharma, and Smith [37]. The intra-task aspect, on the other hand, has not been reported in the literature. The hierarchical predictor designs studied here consider the two aspects as parts of one prediction mechanism. Moreover, as this work shows in the evaluation, both the inter-task and intra-task aspects of hierarchical prediction are important factors in terms of processor performance.

The components used to design the inter-task and intra-task predictors may be taken from existing technology, though modifications and different organizations are necessary to adapt them for use in the hierarchical predictor of a multiscalar processor. To show this adaptation, the hierarchical predictor design is considered in two parts. First, the basic control flow prediction concepts and components are discussed with respect to the schemes studied in this work. Second, the modifications necessary to adapt the schemes for use in the inter-task predictor or the intra-task predictors of a multiscalar processor are described.

5.5.1 Control Flow Prediction

A comprehensive control flow prediction scheme provides a means of generating addresses in order to allow a processor to move through a program control flow graph in a speculative manner. The basic components of such a scheme are shown in Figure 5.4. A supplied address is provided to an index mechanism which constitutes the first level of prediction. The index produced by this mechanism is used to access the prediction state which makes up the second level of prediction. An entry (or possibly several entries depending on the organization of the prediction state) driven by a simple automaton performs a prediction based on target, address, and type information maintained as part of the prediction state. Each of these major aspects of control flow prediction are described in the text that follows.

Before moving on to this description, it is worth pointing out that control flow prediction

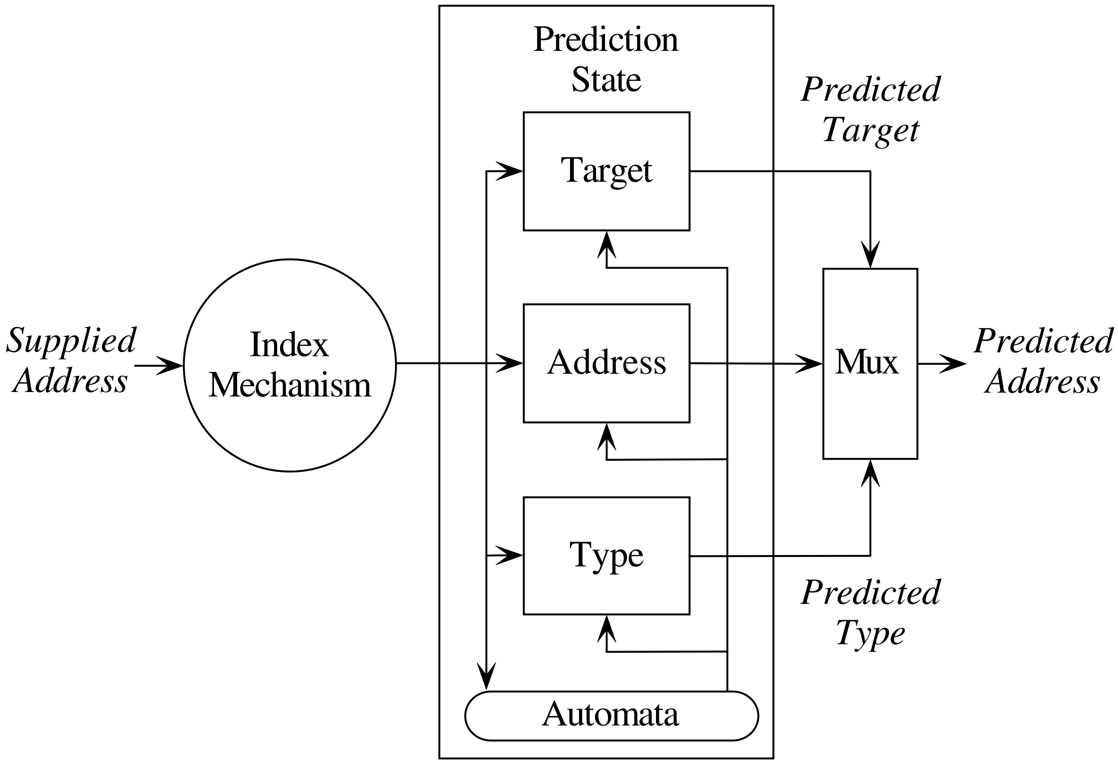


Figure 5.4: Basic control flow prediction components.

has its basis in branch prediction. (The principal distinction is that control flow prediction predicts addresses not just the directions of branches.) One of the first studies of branch prediction techniques, performed by Smith [80], investigated using static heuristics as well as simple automata to select between taken and not taken paths of conditional branches. Branch prediction later evolved into so-called two-level branch predictors as proposed by Yeh and Patt [103] as well as Pan, et al [66]. These designs are widely considered to be the most accurate forms of branch prediction and have been implemented in some form in almost all major microprocessors.

5.5.2 Index Mechanism

The first level of control flow prediction is the index mechanism that provides an index into the second level. The index mechanism is a hashing function of some kind, simple or complex, that takes a control flow stream, divides it into sub-streams, and attempts to direct each sub-stream to a predictor that best captures its behavior. A number of basic strategies (with a plethora of variations) have been proposed. Figures 5.5, 5.6, and 5.7 illustrate the best known index mechanisms from the literature. The descriptions that follow review these index mechanisms and discuss the intuition behind them. These mechanisms are described in terms of their basic workings, not their specific implementation details. References to consult for this information are provided where relevant in the text.

5.5.2.1 Address-Based

An address-based (identified as **addr**) index mechanism is a simple hashing function that divides the control flow stream based on the program counter associated with a particular control flow point. An illustration of this approach is given in Figure 5.5. The figure shows **I** bits from the supplied address being XORed with **I** bits from an optional skewing mask to produce an index to the prediction state. This approach, originally described by Smith [80], ideally forms one sub-stream for each static control flow point in the program. The intuition behind this divider is that each control flow point should have its own predictor because the characteristics and past history of this control flow point are a good predictor of its future behavior. More complex approaches that further subdivide the control flow stream (as described below) can provide a finer decomposition to increase predictability.

5.5.2.2 Pattern-Based

A pattern-based index mechanism divides streams based on the patterns of taken and not taken outcomes associated with some number of preceding control flow points. One strategy to divide streams for a particular control flow point based on patterns is to use some number of the last instances of its own execution. This approach, originally described by Yeh and Patt [103], does not exploit any correlation to other control flow points. Instead, it is designed to exploit repeating patterns in the execution of a single control flow point. Hence, this approach is referred to as a self-pattern-based (identified as **spat**) index mechanism.

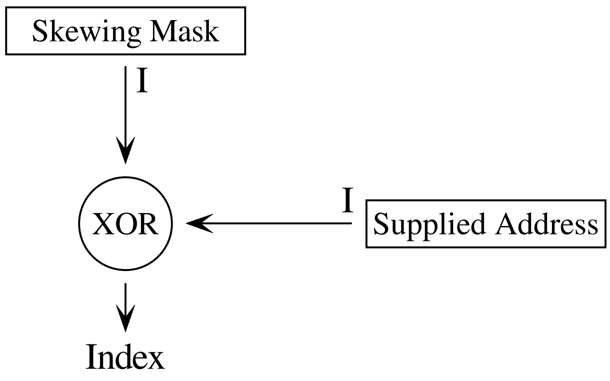


Figure 5.5: Address-based index mechanism.

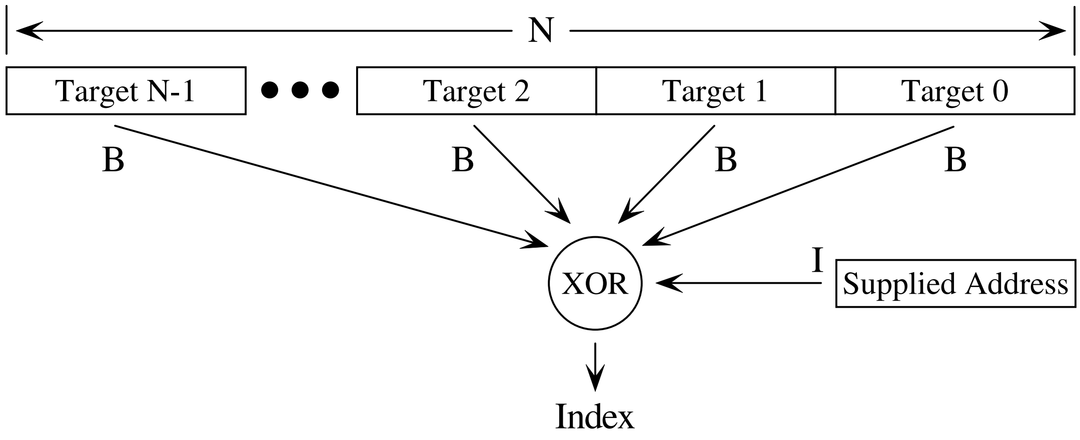


Figure 5.6: Pattern-based index mechanism.

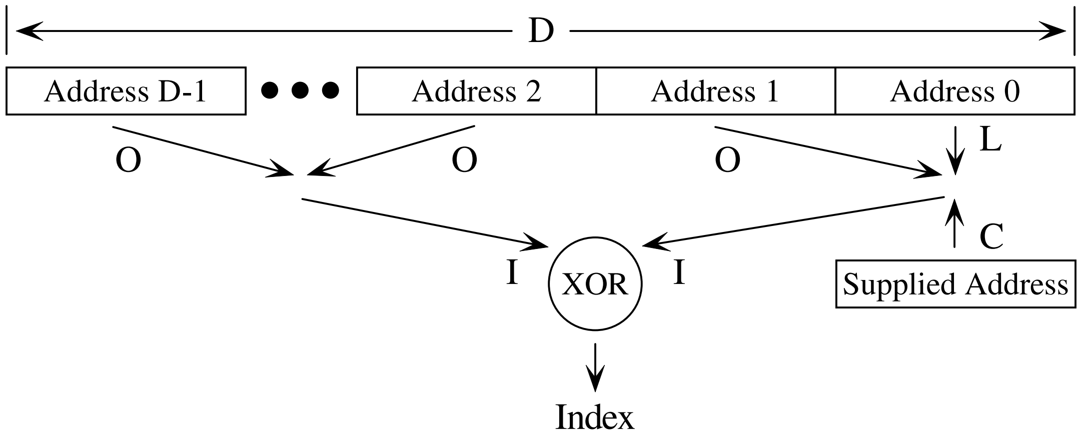


Figure 5.7: Path-based index mechanism.

Another strategy to divide streams for a particular control flow point based on patterns is to use some number of preceding control flow point executions (whether the same or different instructions). This approach, referred to as a global-pattern-based (identified as **gpat**) index mechanism and originally described by Pan, So, and Rahmeh [66], divides streams differently than the self-pattern approach. The intuition behind this approach is that sections of code deal with related information, so control flow points dependent on a particular condition are likely to be placed near control flow points for related conditions. Exploiting the correlation between these conditions may lead to more accurate prediction [104, 106].

An illustration of this approach is given in Figure 5.6. The figure shows **N** target patterns assembled in a pattern register. Each target pattern provides **B** bits, for a total of **I** bits, which are XORed with **I** bits from the supplied address (a technique introduced by McFarling with the gshare predictor [53]) to produce an index to the prediction state . Each successive prediction shifts its target pattern into the pattern register such that the target patterns of the earliest predictions occupy the highest numbered positions in the pattern register. The difference between a self-pattern-based and a global-pattern-based design is whether a single or multiple pattern registers are used. The self-pattern-based strategy uses multiple pattern registers, ideally one for each control flow point. The global-pattern-based strategy uses a single pattern register, one for all control flow points.

5.5.2.3 Path-Based

A path-based (identified as **path**) index mechanism divides streams for a particular control flow point based on the execution path associated with some number of preceding control flow points. A path differs from a pattern because it comprises the program counter addresses of the control flow points, not the directions (taken or not taken) chosen as a result of their execution. This approach was originally studied by Nair [59]. The intuition behind this approach is that the use of program counter addresses of control flow points creates a more accurate representation of program control flow than do execution patterns of control flow point directions. Creating a better representation of program control flow may provide a better association between control flow points and predictors that capture their behavior.

An illustration of this approach, in particular the scheme developed by Jacobson, Bennett, Sharma, and Smith [37], is given in Figure 5.7. The figure shows **D** control flow point addresses assembled in a path register. The supplied address provides **C** bits from the control flow point being considered, the latest address in the path register provides **L** bits, and the other addresses in the path register provide **O** bits. Depending on the total number of bits provided, the collection of bits is separated into a number of parts of **I** bits each and XOR folded that many times to produce an index to the prediction state. Each successive prediction shifts its address into the path register such that the addresses of the earliest predictions occupy the highest numbered positions in the path register.

5.5.3 Prediction State

The second level of control flow prediction is the prediction state that provides the actual prediction. The prediction state comprises target, address, and type information as well as automata to control updates to the prediction state. This collection of prediction state may be organized as a single structure [105] or as multiple structures [14]. The choice, however, is an implementation decision whose impact on overall performance depends on the specifics of the implementation. The purpose of this description is to explain the components of the prediction state in basic terms, so no distinction is made between these two choices.

5.5.3.1 Target

The target component of the prediction state records a target identifier for control flow points that select between two or even several possible control flow paths. In conventional control flow prediction designs this component is used to indicate the taken or not taken path for conditional branches for a prediction and is usually given implicitly by the automata used (discussed below). In a multiscalar control flow prediction design it is used to indicate one of many possible task exits identified by the compiler and is usually given explicitly with the automata used. It is not strictly needed for other control flow types that can only direct control flow along one path as a result of their execution. All indirect forms of control flow do not require this component since these instructions can only follow one control path as a result of their execution, even though each execution may direct control flow to different program locations (*i.e.*, a jump through a register has only one target, but the target may correspond to many program locations depending on the register value).

5.5.3.2 Address

The address component of the prediction state provides the address to which control flow is directed in advance of the actual calculation of this address by the associated control flow instruction. In general, there is some delay between the time a control flow point is encountered and the time the address to which it redirects control is computed. Keeping the address associated with prior behavior of a control flow point allows expeditious prediction of its control flow address when it is encountered again. For most control flow types, the address is just maintained in a structure that is accessed as needed. However, for procedure returns, the address is best maintained in a return address stack since this structure captures the behavior of procedure calls and returns with much greater accuracy [42] than a branch target buffer.

5.5.3.3 Type

The type component of the prediction state allows the control flow prediction to differentiate between different types of control flow points. Because different types of control flow exhibit different types of behavior (e.g. a conditional branch as compared to a procedure return), no single strategy suffices for all cases. Knowing the type of the control flow to be predicted allows the appropriate action to be taken during control flow prediction. For instance, knowing that the control flow to be predicted is a procedure return may allow the predictor to choose the return address stack instead of the branch target buffer as the source for a predicted address. The type information must usually be provided before the control flow point is encountered to be of profitable use. This type information may maintained in a structure or determined in some other more speculative fashion.

5.5.3.4 Automata

The automata component of the prediction state in a sense drives the other three components since it directs updates to the other prediction state components. The most widely used automaton is the saturating up/down counter. The counter is incremented (not beyond its maximum value) on correct predictions and is decremented (not beyond its minimum value) on incorrect predictions. Above some threshold counter value, the confidence in the existing information in the prediction state is strengthened. Below this threshold, the confidence in the existing information is weakened and at some point replaced with new information.

5.5.4 Multiscalar Considerations

A hierarchical predictor (one using multiple program counters) may be designed from sequential predictors (ones using a single program counter) in a straightforward manner. The inter-task portion of the predictor consists of a sequential predictor based on one of the designs described above. Likewise, each intra-task portion of the predictor consists of such a sequential predictor as well. Each sequential predictor is in turn modified in a few simple respects (as described below) to accommodate specific differences from its old role as an individual sequential predictor to its new role as part of a hierarchical predictor.

The main difference in these roles is that the number and sequence of control flow points, as seen by the index mechanism and prediction state of the predictor, are changed. Each predictor now has an incomplete view of the control flow that led to the control flow point that must be predicted. In the case of inter-task predictions, there are gaps (possibly large) in the observed control flow points. In the case of intra-task predictions, only short sequences of control flow points are seen. In addition, because there are multiple intra-task predictors, the gaps between these short sequences may contain a considerable number of control flow points.

5.5.4.1 Inter-Task Predictor

The existence of gaps in observed control flow points (because of arbitrary internal task control flow) means that the inter-task predictor must accommodate multiple targets for each control flow point predicted. With respect to traditional prediction, multiple targets implies more than two. The hierarchical predictors studied in this thesis handle at most four targets. The changes to the index mechanism to support multiple targets are minor for pattern-based designs and only involve increasing the number of bits used to represent a target pattern in the pattern register and the distance of the shift when a target is put into the pattern register. The index mechanisms for path-based and address-based designs require no changes.

The target component of the prediction state must likewise support an increased number of bits for the target identifier. In addition, the address and type components must be replicated for each of the multiple targets. Because the multiscalar processor organizations studied in this thesis rely on the compiler to provide the type and address information in the task header inserted directly into the program binary, this information is maintained in a separate task cache (already described) rather than in the prediction state. The prediction state does, however, still maintain addresses for indirect control flow (which cannot be provided by the compiler) and a return address stack for function returns. The automata component does not require any change.

5.5.4.2 Intra-Task Predictor

The intra-task predictor itself does not require any change to its index mechanism or prediction state since the control flow predictions it performs are similar to those it was intended for in conventional processors. Nevertheless, because only short sequences of control flow points are seen by the intra-task predictor, it is difficult to achieve any correlation with preceding control flow points, whether the same or different instructions; these preceding control flow points are seen by the inter-task predictor or by another intra-task predictor. Without correlation of some kind, an intra-task predictor that uses a sophisticated index mechanism may not work as well as intended or in the worst case may perform worse than if a simple index mechanism is used. A means of addressing this issue is to incorporate some inter-task information into the intra-task information in hopes of achieving a profitable degree of correlation.

A basic approach that is applicable to all combinations of inter-task and intra-task index mechanisms is to use the index produced by the inter-task index mechanism as an additional input to the XOR operation performed by the intra-task index mechanism. This approach offers only a limited amount of correlation, though, since the inter-task information does not really become a part of the intra-task information. Nonetheless, this approach is the only option, unless both the inter-task and the intra-task predictors use correlation based on the behavior of the most recent preceding control flow points, as path-based and global-pattern-based approaches do. The address-based and self-pattern-based approaches do not maintain control flow history in a way that it can be incorporated into an intra-task predictor or in a way that it can be incorporated into from an inter-task predictor.

Using just a combination of path-based and/or global-pattern-based predictors, though, a more sophisticated approach incorporating inter-task information into the intra-task information may be constructed. In the straightforward cases where the inter-task and intra-task predictors are of the same type, the inter-task path/pattern register information (from the time the prediction for a task is made) is copied into the intra-task path/pattern register associated with the execution of the task. Thus, the intra-task predictor simply appears to be a continuation of the inter-task predictor. In the more complex cases where the inter-task and intra-task predictors are not of the same type, these approaches are not always applicable. Specifically, an inter-task global-pattern-based predictor cannot provide enough information to fill the path register for an intra-task path-based predictor. Ironically, an inter-task path based predictor provides too much information to fill the pattern register for an inter-task global-pattern-based predictor. However, the index produced by the path-based predictor may be used instead to fill the global-pattern-based predictor.

5.6 Hierarchical Predictor Evaluation

The hierarchical predictor evaluation is divided into three parts to better understand the individual as well as the collective influence of hierarchical predictor design factors.

First, an inter-task study is performed to focus on the extent to which inter-task prediction affects the execution of the processor. Second, an intra-task study is performed to focus on the extent to which intra-task prediction affects the execution of the processor. For both studies, the evaluation involves varying the prediction schemes for the inter-task and intra-task predictors to explore the range of behavior with the path-based, global-pattern-based, self-pattern-based, and address-based alternatives described earlier.

To separate inter-task from intra-task effects, each of these aspects of hierarchical prediction is studied in isolation of the other by using a perfect prediction mechanism for the aspect not under consideration. Thus, for the inter-task portion of the study, a perfect mechanism is used for intra-task prediction, and similarly for the intra-task portion of the study, a perfect mechanism is used for inter-task prediction. For the most part, the inter-task and intra-task predictors are independent of one another, so this approach does reflect the expected characteristics of the predictors when combined.

The third part of the evaluation puts together the results of the inter-task and the intra-task studies to select the best combination of schemes, matching the best inter-task predictor with the best intra-task predictor, for further evaluation. In particular, this part studies the behavior of the predictor overall and its impact on processor performance (with no perfect mechanisms of any kind). In addition, this part of the evaluation examines the effect of the amount of predictor state on the behavior of the predictors by varying the amount devoted to each type of predictor for a full range of inter-task and intra-task designs.

5.6.1 Metrics

The metric of most significance for the hierarchical predictor (as for any processor component) is its impact on processor performance. As a result, the actual processor performance given as the harmonic mean (HMEAN) of instructions per cycle (discussed in Section 4.6.1) is provided for this evaluation. Even so, it is still often useful to consider a non-performance oriented metric to compare the behavior of different predictors. Thus, this evaluation also provides the misprediction ratio – the number of mispredictions per prediction performed by the predictor – of the various hierarchical predictor configurations studied.

The misprediction (and the prediction) counts are based on those observed for the useful tasks and the useful instructions from a program execution run. Any predictions beyond a misprediction are not counted as a part of the misprediction ratio. This misprediction ratio is given as the unweighted arithmetic (as opposed to the harmonic or geometric) mean (AMEAN) for each of the two groups of benchmarks, SPEC CINT95 and SPEC CFP95, because the misprediction ratio is proportional to the program execution time; it should not be weighted because each benchmark program is meant to be representative of a particular type of (integer or floating point) application independent of the number of instructions it executes for its particular input.

5.6.2 Inter-Task Study

This part of the hierarchical predictor evaluation isolates the effect of the inter-task predictor from the intra-task predictor by using an intra-task predictor which provides perfect prediction accuracy. Since only the hierarchical predictor is under consideration, the other components of a multiscalar processor are held constant as given by the standard configuration shown in Table 3.11 of Chapter 3.

5.6.2.1 Configurations

One configuration from each of the four strategies described earlier are considered for the index mechanism of the inter-task predictor. The amount of the prediction state is not varied in this study, but will be in the overall consideration of the hierarchical predictor. Thus, for this part of the evaluation, each index mechanism produces a 16-bit index for the 64K-entry prediction state (described below); that is, **I**=16 from Figures 5.5, 5.6, and 5.7. In all descriptions below, an address is shifted right 2 bits (since addresses are word aligned, and this portion of the address provides no useful information).

For the address-based, **addr**, design, the low order 16 bits of the supplied address are used as an index; no skewing mask is used, so no XOR operation is performed. For the pattern-based designs, the pattern register contains **N**=8 target patterns with **B**=2 bits from each pattern for a total of 16 bits. For the self-pattern-based, **spat**, design, 1K pattern registers are provided and indexed with the low order 10 bits of the supplied address; no XOR operation is performed. For the global-pattern-based, **gpat**, design, one pattern register is provided and XORed with the low order 16 bits of the supplied address. For the path-based, **path**, design, the path register contains **D**=7 addresses with **L**=6 bits from the latest address and **O**=3 bits from other addresses. The supplied address provides **C**=8 bits from the control flow point being considered. This configuration produces a total of 32 bits which are split into two parts of 16 bits and XORed.

Much of the inter-task control flow information is provided by the compiler in the form of task descriptors inserted into the code. These task descriptors are held in the task cache, which maintains addresses and a type specifier for each of the four possible task targets. A common task cache configuration of 1024 entries with 64 byte task descriptors using 2-way associativity and least recently used replacement supports the evaluation of each predictor type. The task cache eliminates the need to maintain addresses for non-indirect control flow and types for all control flow in the inter-task prediction state. The rest of inter-task control flow information is maintained as part of the prediction state. A common 64K-entry inter-task prediction state configuration accessed via the index provided is used to evaluate each predictor type. Each entry of the inter-task prediction state contains a 2-bit predicted target identifier for one of the four possible targets, a predicted address for indirect control flow (aligned, so minus the two low-order bits), and a 2-bit saturating up/down counter with predicted target or address replacement occurring upon update if the counter is zero (roughly 34 bits per entry).

5.6.2.2 Results

Figures 5.8 and 5.9 give the inter-task misprediction ratios for different prediction schemes for the SPEC CINT95 and the SPEC CFP95 benchmarks, respectively. All of the predictors provide nearly the same prediction accuracy regardless of the processing unit configuration studied, though the **spat** design seems to have somewhat higher accuracy (fewer mispredictions) with smaller numbers of tasks. (This effect is related to tasks, not processing units, since the 4 processing unit designs running multiple tasks suffer degraded accuracy.)

For the integer benchmarks, **path** is almost always the most accurate design, but **gpat** is nearly as accurate. The other designs, **addr** and **spat**, are consistently less accurate (more mispredictions), sometimes by a significant amount, 0.12 compared to 0.06 mispredictions per prediction. For the floating point benchmarks, the **addr** design stands out as being less accurate than the other designs, though the absolute amount is only about 0.04 compared to 0.02 mispredictions per prediction. The **path** and **gpat** designs have the same behavior as for the integer benchmarks. However, the **spat** design sometimes performs somewhat better or somewhat worse than these design depending on the number of tasks.

Figures 5.10 and 5.11 give the performance in instructions per cycle with each of the different inter-task predictors for the SPEC CINT95 and the SPEC CFP95 benchmarks, respectively. In addition to each of the predictor designs studied, the performance using a perfect inter-task prediction mechanism, **perf**, is shown. For the floating point benchmarks, the performance is nearly the same regardless of the predictor design used, even for the perfect predictor. In contrast, for the integer benchmarks, performance correlates with the misprediction ratios. Yet, the impact of the prediction accuracy is clearly dependent on the number of tasks that run in parallel as well as the policy with which the tasks are run (for the configurations that support multiple tasks per processing unit).

In particular, the gap between the performance using predictors of different accuracy increases as the number of tasks run in parallel increases. The 2 processing unit configuration (**m2**) shows little difference in performance between the more and less accurate predictors, even for the perfect predictor. However, the 16 processing unit configuration (**m16**) shows a difference in performance of 0.50 instructions per cycle or 17% between the more and less accurate predictors, with an even greater difference for the perfect predictor of 1.50 instructions per cycle or 50%. The same kind of performance difference is seen in going from 2 to 4 tasks running per processing unit and in going from running these tasks one-at-a-time to all-at-once.

5.6.3 Intra-Task Study

As in the inter-task study, the effect of the intra-task predictor is isolated from the inter-task predictor. This isolation is accomplished in the same way as for the inter-task; that is, the inter-task predictor is made to have perfect prediction accuracy, while the various alternatives for the intra-task predictor are evaluated. Again, as in the inter-task study, the standard configuration shown in Table 3.11 of Chapter 3 is assumed for the other components of a multiscalar processor since only the hierarchical predictor is under consideration here.

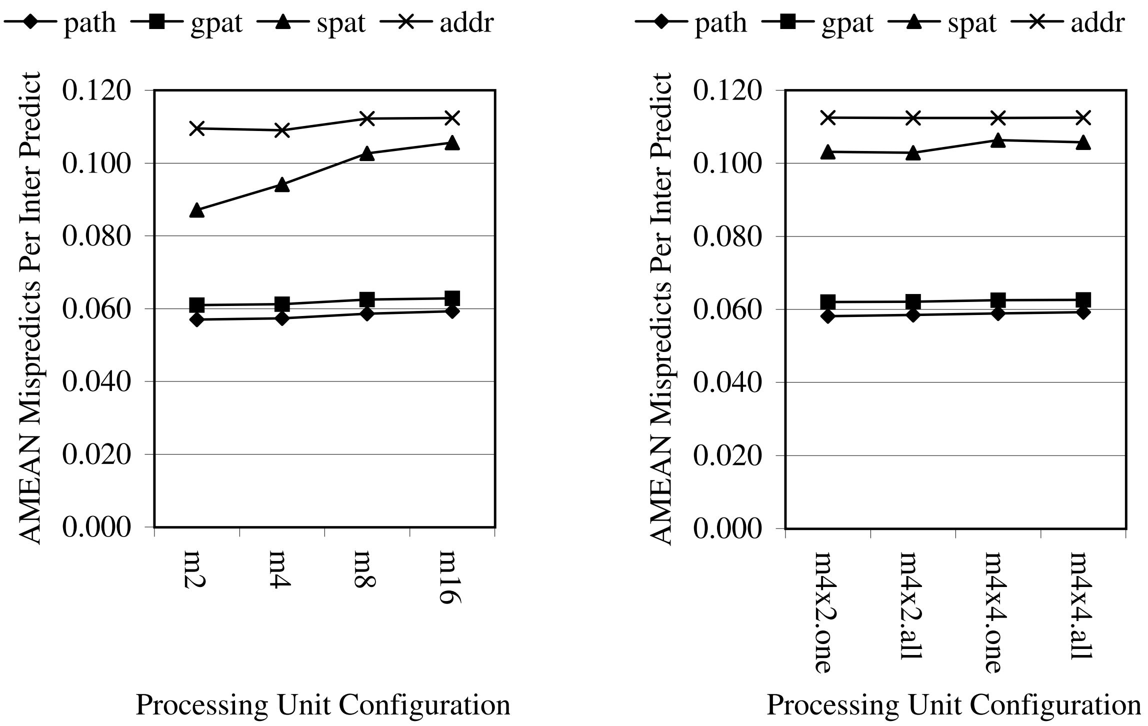


Figure 5.8: SPEC CINT95 inter-task misprediction ratio for different inter-task prediction schemes.

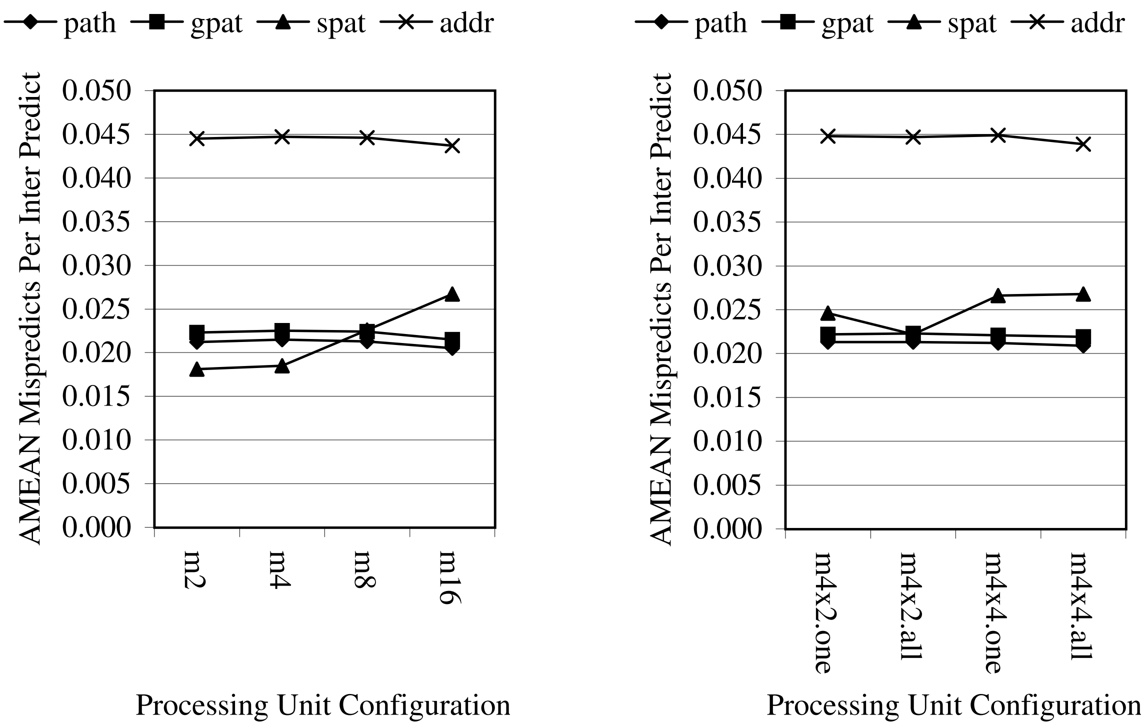


Figure 5.9: SPEC CFP95 inter-task misprediction ratio for different inter-task prediction schemes.

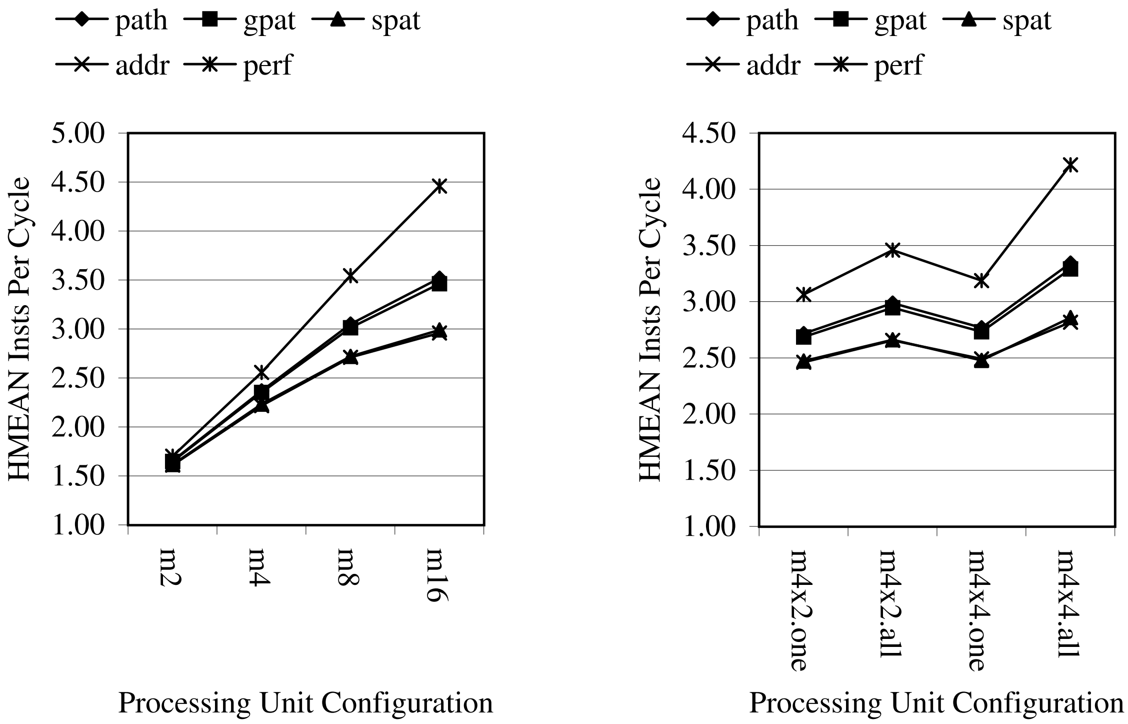


Figure 5.10: SPEC CINT95 performance for different inter-task prediction schemes.

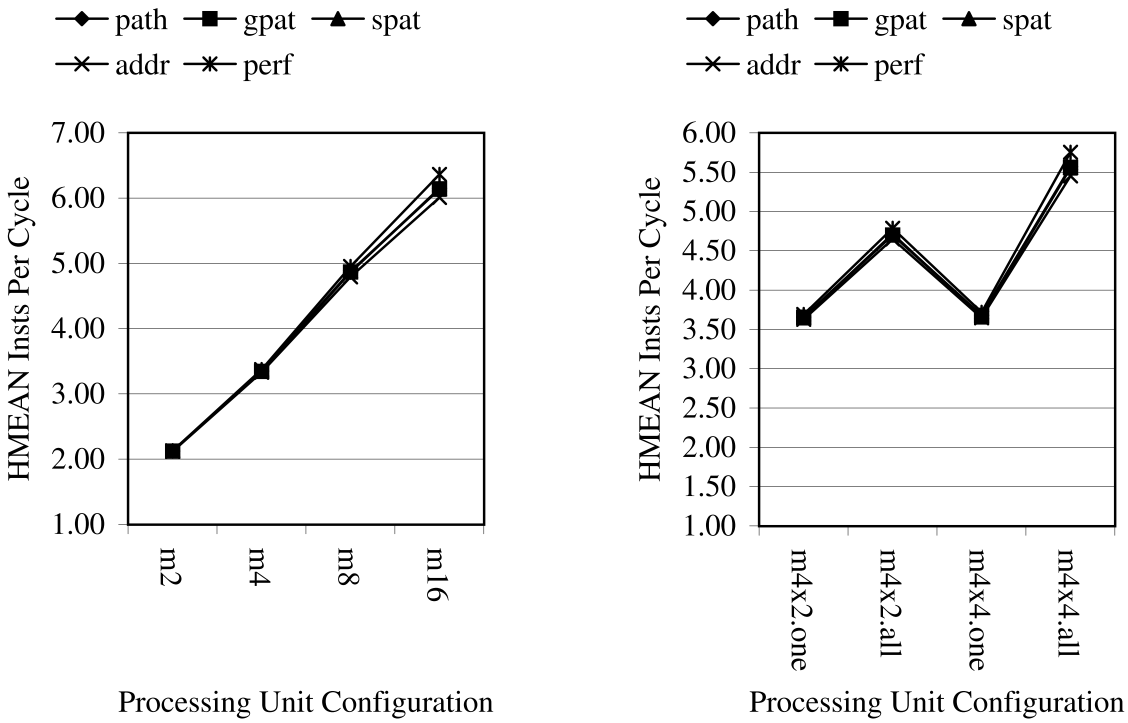


Figure 5.11: SPEC CFP95 performance for different inter-task prediction schemes.

5.6.3.1 Configurations

One configuration from each of the four strategies described earlier are considered for the index mechanism of the intra-task predictor. The amount of prediction state for each intra-task predictor is not varied in this intra-task study (just as it is not in the inter-task study). For this part of the evaluation, each index mechanism produces a 16-bit index for the 64K-entry prediction state (described below); that is, **I**=16 from Figures 5.5, 5.6, and 5.7. In all descriptions below, an address is shifted right 2 bits (since addresses are word aligned, and this portion of the address provides no useful information). All predictors are configured in the same manner as described for the inter-task study except for the differences described below.

The pattern register for the pattern-based designs holds twice as many target patterns with half as many bits from each pattern because intra-task target identifiers are only a single bit (taken or not taken); in the terminology used in the inter-task description, the pattern register contains **N**=16 target patterns with **B**=1 bit from each pattern for a total of 16 bits. As described earlier, inter-task information is incorporated into the intra-task information to improve control flow correlation. Since the inter-task control flow information is provided by a perfect prediction mechanism for the intra-task study, it is assumed that the path register from the path-based design is used to hold the control flow addresses of all recent control flow points.

The inter-task information is used in conjunction with all of the intra-task predictors. For the address-based, **addr**, and self-pattern-based, **spat** designs, the index provided by the inter-task index mechanism is XORed with the index provided by the intra-task index mechanism. For the global-pattern-based, **gpat**, design, the index provided by the inter-task index mechanism is written into the intra-task pattern register when the task is assigned. For the path-based, **path**, design, the inter-task path register is written into the intra-task path register when the task is assigned.

None of the intra-task control flow information is provided by the compiler as for the inter-task control flow. Thus, all intra-task control flow information is maintained within the prediction state. A 64K-entry intra-task prediction state configuration accessed via the index provided is used to evaluate each predictor type. Each entry of the intra-task prediction state contains a 1-bit predicted target identifier for one of two possible targets (for conditional branches), a predicted address (aligned, so minus the two low-order bits), an encoded control flow type field, and a 2-bit saturating up/down counter with predicted target or address replacement occurring upon update if the counter is zero (roughly 35 bits per entry).

5.6.3.2 Results

Figures 5.12 and 5.13 give the intra-task misprediction ratios for different prediction schemes for the SPEC CINT95 and the SPEC CFP95 benchmarks, respectively. All of the predictors provide somewhat lower prediction accuracy (more mispredictions per prediction) as more processing units are used, except for the case of 2 processing units (**m2**) which goes against this trend. The **spat** design is particularly influenced by this effect, with the prediction accuracy changing from 0.10 or 0.25 mispredictions per prediction for integer and from 0.02 to

0.06 mispredictions per prediction for floating point in going from 2 to 16 processing units, while the **path**, **gpat**, and **addr** designs change by less than 0.01 mispredictions per prediction (in absolute terms) for both integer and floating point. (Unlike the inter-task case, this effect in this intra-task case is related to processing units not tasks, since the 4 processing unit designs running multiple tasks do not suffer degraded accuracy.)

For the integer benchmarks, **gpat** is almost always the most accurate design, but **path** and **addr** are nearly as accurate. The **spat** design is consistently less accurate, sometimes by a significant amount, nearly as much as 0.20 mispredictions per prediction. For the floating point benchmarks, The **path** and **gpat** designs have the same behavior as for the integer benchmarks. However, the **addr** design has somewhat lower prediction accuracy, about 0.01 mispredictions per prediction, compared to these two for the floating point benchmarks though it had comparable accuracy on the integer benchmarks. The **spat** is consistently less accurate than the other designs, but not as much so on the floating point as on the integer programs.

Figures 5.14 and 5.15 give the performance in instructions per cycle with each of the different intra-task predictors for the SPEC CINT95 and the SPEC CFP95 benchmarks, respectively. In addition to each of the predictor designs studied, the performance using a perfect intra-task prediction mechanism, **perf**, is shown. As for the inter-task predictor, the performance on the floating point benchmarks is nearly the same regardless of the intra-task predictor design used, even for the perfect predictor. As before, for the integer benchmarks, performance correlates with the misprediction ratios. While the impact of the prediction accuracy is clearly dependent on the number of tasks that run in parallel as well as the policy with which the tasks are run (for the configurations that support multiple tasks per processing unit), it much less so for intra-task than for inter-task predictors.

Just as for the inter-task predictor, the gap between the performance using intra-task predictors of different accuracy increases as the number of tasks run in parallel increases. The 2 processing unit configuration (**m2**) shows little difference in performance between the more and less accurate predictors, even for the perfect predictor. However, the 16 processing unit configuration (**m16**) shows a difference in performance of 0.50 instructions per cycle or 13% between the more and less accurate predictors, with an even greater difference for the perfect predictor. The same kind of performance difference is seen in going from 2 to 4 tasks running per processing unit and in going from running these tasks one-at-a-time to all-at-once, but it is a very small difference. Furthermore, the difference overall is much smaller for intra-task than for inter-task prediction, even though the prediction accuracy between more and less accurate predictors is much larger, 0.15 to 0.20 mispredictions per prediction for intra-task predictors and only 0.06 mispredictions per prediction for inter-task predictors.

5.6.4 Overall

Overall, the inter-task and intra-task studies indicate that performance is more sensitive to the prediction accuracy the more aggressive the processor design used (as far as the number of processing units, the number of tasks, and/or the policy for running multiple tasks per processing unit). Moreover, performance is more sensitive to the prediction accuracy of the

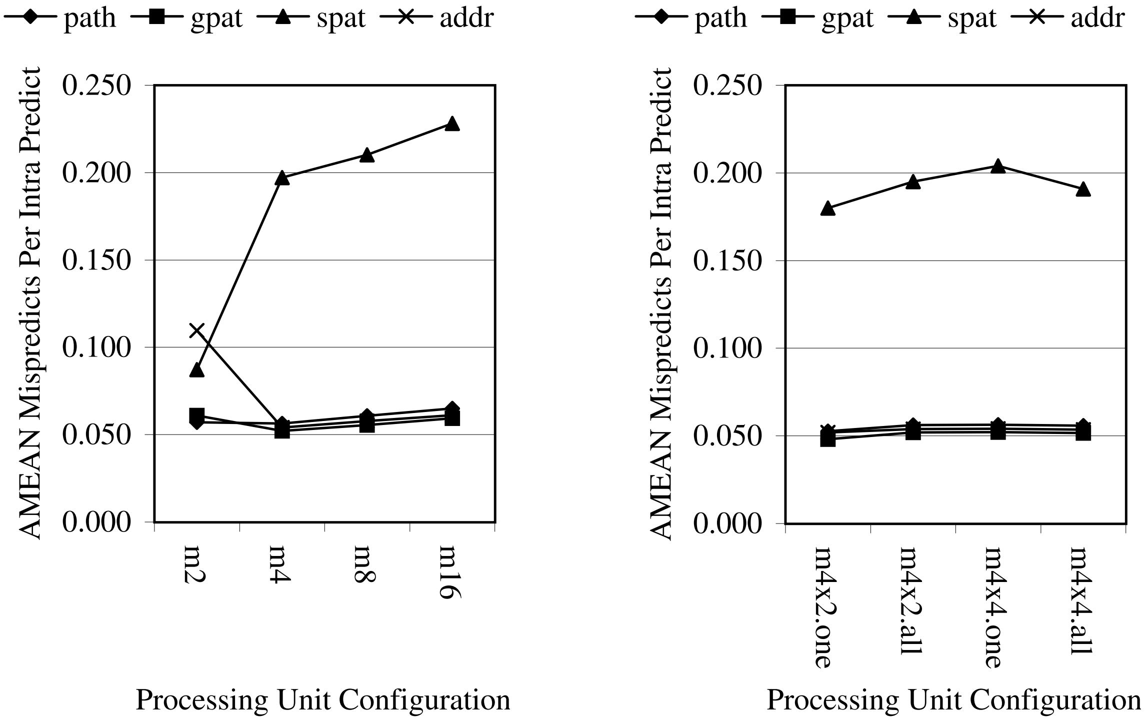


Figure 5.12: SPEC CINT95 intra-task misprediction ratio for different intra-task prediction schemes.

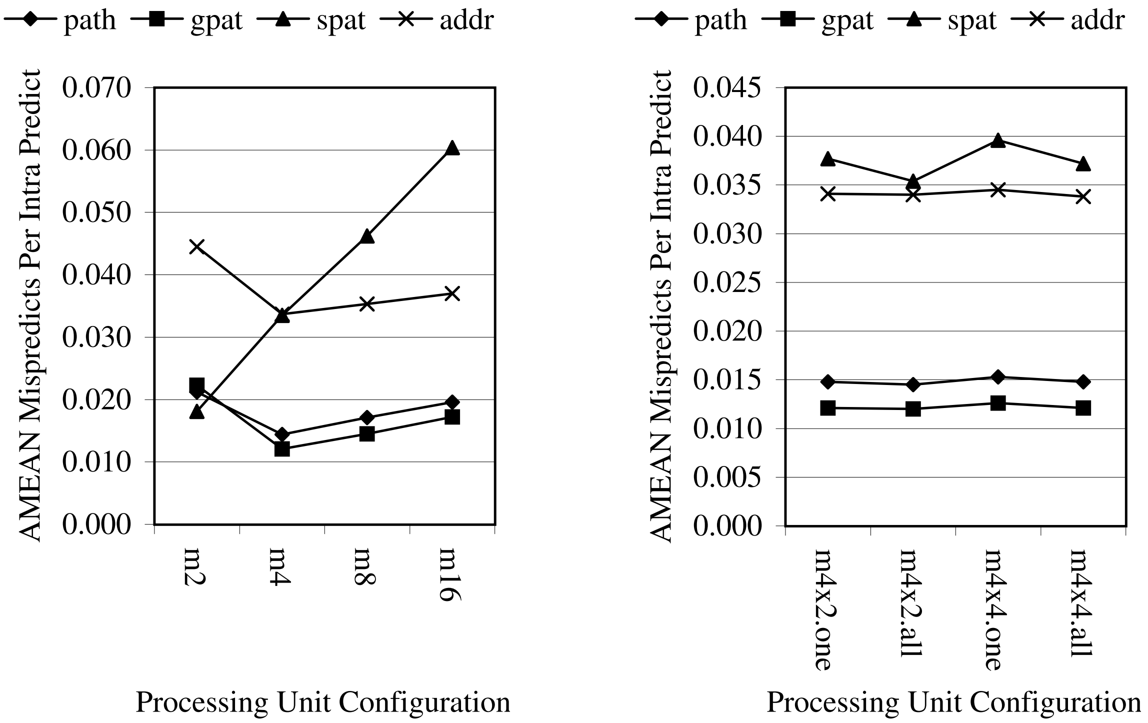


Figure 5.13: SPEC CFP95 intra-task misprediction ratio for different intra-task prediction schemes.

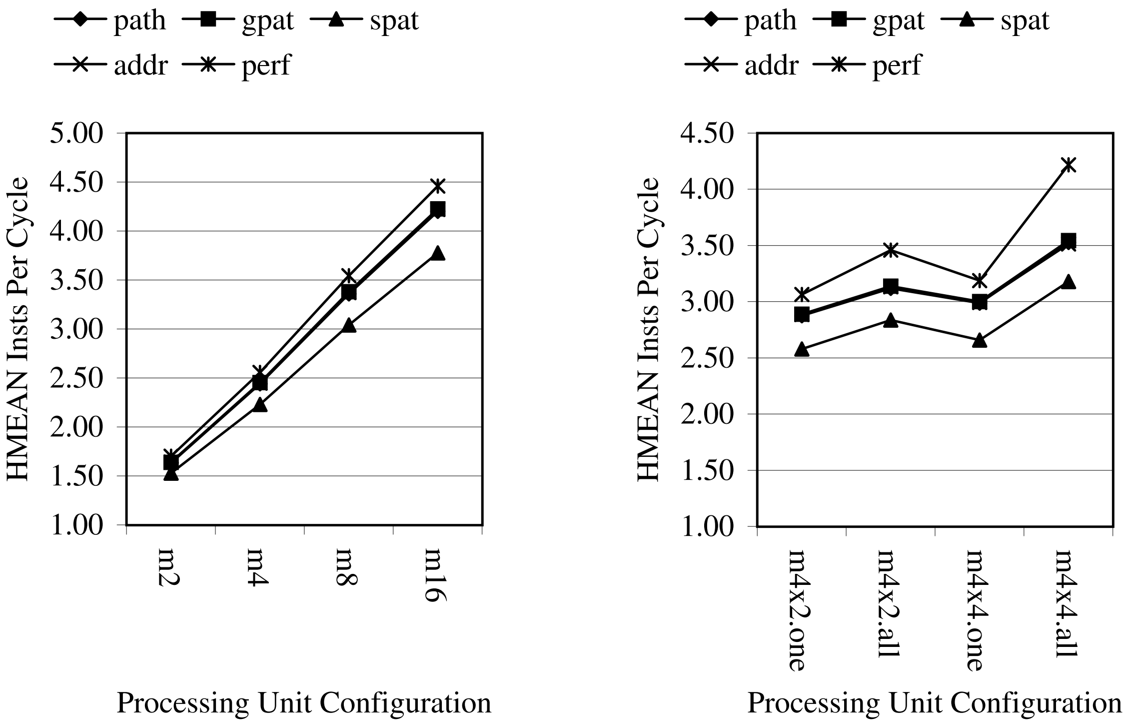


Figure 5.14: SPEC CINT95 performance for different intra-task prediction schemes.

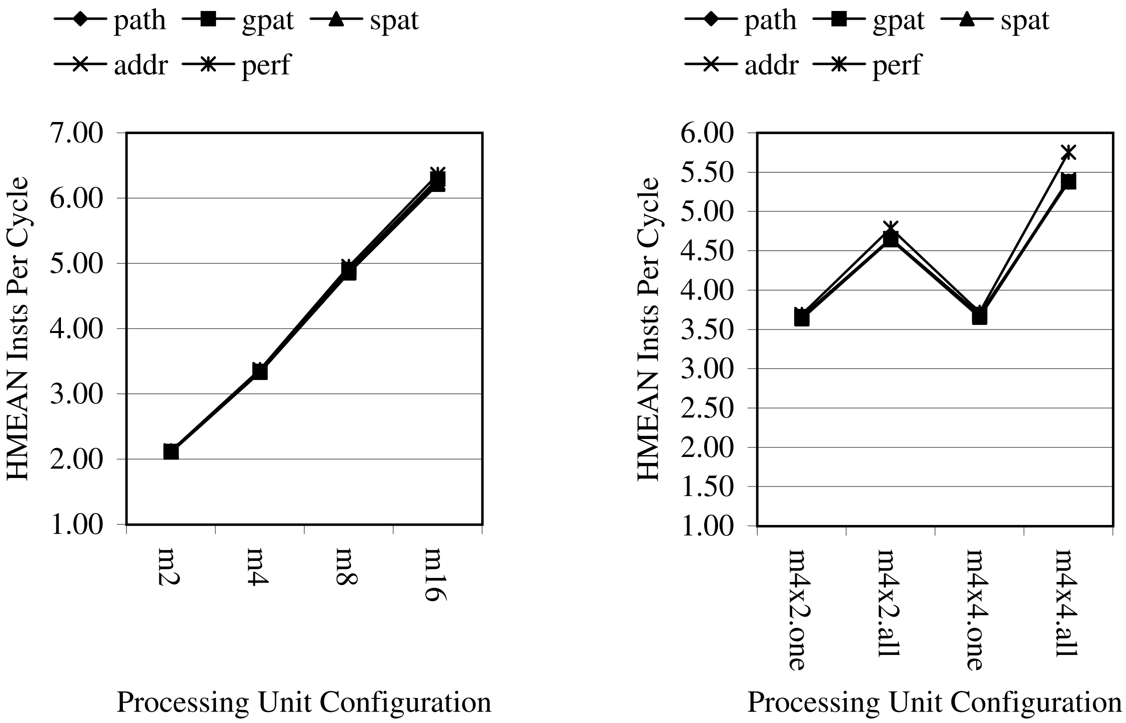


Figure 5.15: SPEC CFP95 performance for different intra-task prediction schemes.

inter-task predictor than the intra-task predictor. This behavior is consistent with the fact that parallelism is exposed among processing units by the inter-task predictor and within an individual processing unit by the intra-task predictor.

Up to this point, the hierarchical predictor has not been evaluated with real mechanisms for both the inter-task and the intra-task prediction. In the earlier studies, a perfect mechanism was used for the aspect not under consideration. Sometimes in the studies there are several predictors that perform nearly as well. Nonetheless, for the purpose of an overall evaluation, only one predictor type is chosen for each aspect of hierarchical prediction. For the inter-task aspect, the **path** predictor is chosen as the best mechanism. For the intra-task aspect, the **gpat** predictor is chosen as the best mechanism.

5.6.4.1 Configurations

Using the combination of a path-based inter-task predictor and global-pattern-based intra-task predictors chosen from the results of the inter-task and intra-task studies, a range of a hierarchical predictor designs may be evaluated. In particular, the amount of prediction state that is devoted to each aspect of hierarchical prediction is varied to study the effect that this parameter of the predictor has on prediction accuracy as well as overall performance. Recall that for the inter-task and intra-task studies, all predictor designs used a 64K-entry prediction state for each predictor, where an inter-task entry was roughly 34 bits and an intra-task entry was roughly 35 bits. For this part of the evaluation, the prediction state for the inter-task predictor as well as the intra-task predictors is varied among 8K-entry, 16K-entry, 32K-entry, and 64K-entry to evaluate hierarchical predictors that comprise the cross-product of these sizes (a total of 16 different configurations).

The inter-task and intra-task predictors are configured similarly as described in the earlier studies. For the path-based, **path**, inter-task design, the path register is maintained exactly as described in the inter-task study (**D**=7, **L**=6, **O**=3, **C**=8). The index is produced in the same way as well to provide 16 bits with which to access the prediction state. However, only the number of bits needed for the amount of state are used for an index. The 8K-entry prediction state uses the low 13 bits, the 16K-entry prediction state uses the low 14 bits, the 32K-entry prediction state uses the low 15 bits, and the 64K-entry prediction state uses all 16 bits. For the global-pattern-based **gpat** intra-task design, the pattern register and index are provided exactly as described for the intra-task study. Again, though, only the number of bits needed for the amount of prediction state, as described above for the inter-task index, are used for the intra-task index.

5.6.4.2 Inter-Task Results

The inter-task results are gathered in two steps. First, to study different inter-task predictor sizes for a range of processing unit configurations, the results focus on designs where the intra-task prediction state is fixed to 64K-entry, while the inter-task prediction state is varied as 8K-entry, 16K-entry, 32K-entry, and 64K-entry. Second, once the trends for different processing configurations have been established, the results focus on one processing unit configuration, 16 processing units, to study different inter-task predictor sizes in conjunction with different intra-task predictor sizes.

Figures 5.16 and 5.17 give the misprediction ratio and the performance in instructions per cycle, respectively, with each of the different inter-task predictor sizes for the SPEC CINT95 benchmarks. The misprediction ratio increases steadily as the inter-task predictor size is halved. However, the magnitude of this difference is not that significant. Comparing the smallest (8K-entry) to the largest (64K-entry) predictor, the difference is less than 0.012 mispredictions per prediction. Moreover, the overall performance shows relatively insignificant differences between the different predictor sizes, though the differences do increase as the number of tasks in concurrent execution is increased (as described earlier).

Figures 5.18 and 5.19 focus on the 16 processing unit configuration for the SPEC CINT95 results just described. In the earlier results, the intra-task prediction state was fixed at 64K-entry. Now, the intra-task prediction state is varied among 8K-entry, 16K-entry, 32K-entry, and 64K-entry as well. The inter-task misprediction ratio is not influenced by the changes to the intra-task predictor size, as expected if the two types of prediction are more or less independent. However, the overall performance does change because it is dependent on the behavior of both predictors. As shown in the figure, there is a steady increase in performance as the intra-task predictor size is increased which is fairly constant as the inter-task predictor size is increased. Overall, the performance difference between the smallest (8K-entry inter-task and intra-task predictors) and the largest (64K-entry inter-task and intra-task predictors) combined designs is around 0.20 instructions per cycle or somewhat less than a 10% increase.

Figures 5.20 and 5.21 give the misprediction ratio and the performance in instructions per cycle, respectively, with each of the different inter-task predictor sizes for the SPEC CFP95 benchmarks. The misprediction ratio does not show significant and/or consistent differences as the inter-task predictor size is halved. Note that, for the scale of the graph, the magnitude of these differences is only around 0.001 mispredictions per prediction comparing the smallest (8K-entry) to the largest (64K-entry) predictors. Nonetheless, there does seem to be an overall trend that the misprediction ratio decreases as the number of tasks in concurrent execution. Looking at the overall performance, there is no distinguishable differences between the different predictor sizes.

Figures 5.22 and 5.23 focus on the 16 processing unit configuration for the SPEC CFP95 results just described. In the earlier results, the intra-task prediction state was fixed at 64K-entry. Now, the intra-task prediction state is varied among 8K-entry, 16K-entry, 32K-entry, and 64K-entry as well. The inter-task misprediction ratio is not influenced by the changes to the intra-task predictor size, as expected if the two types of prediction are more or less independent. Note again that, for the scale of the graph, the magnitude of these differences is less than 0.0005 mispredictions per prediction comparing the smallest (8K-entry) to the largest (64K-entry) inter-task predictors. Though, the overall performance is dependent on the behavior of both predictors, it does not change much here because the intra-task prediction accuracy does not change much for different intra-task predictor sizes (as discussed next).

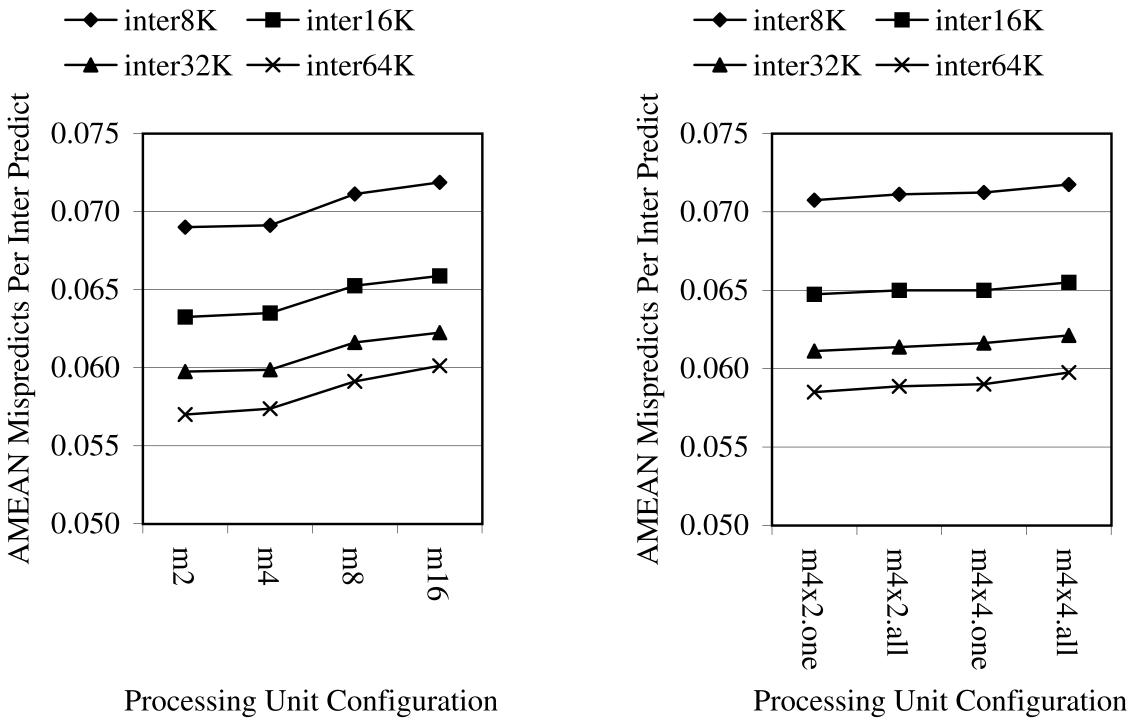


Figure 5.16: SPEC CINT95 inter-task misprediction ratio for different inter-task predictor sizes.

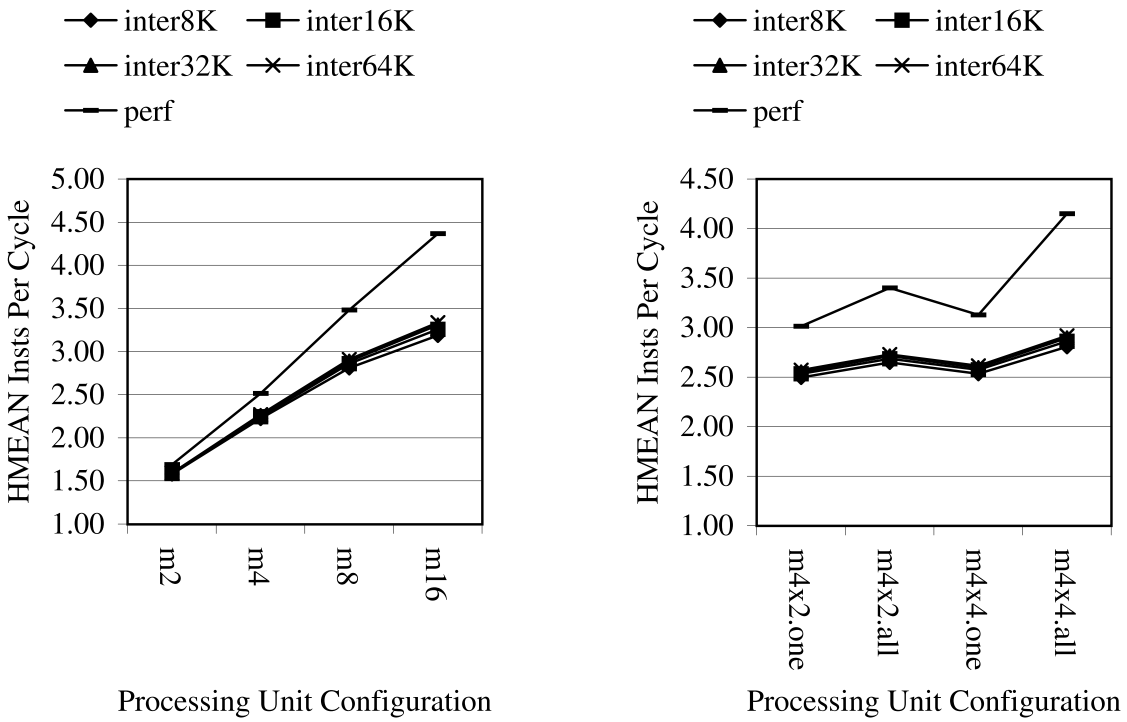


Figure 5.17: SPEC CINT95 performance for different inter-task predictor sizes.

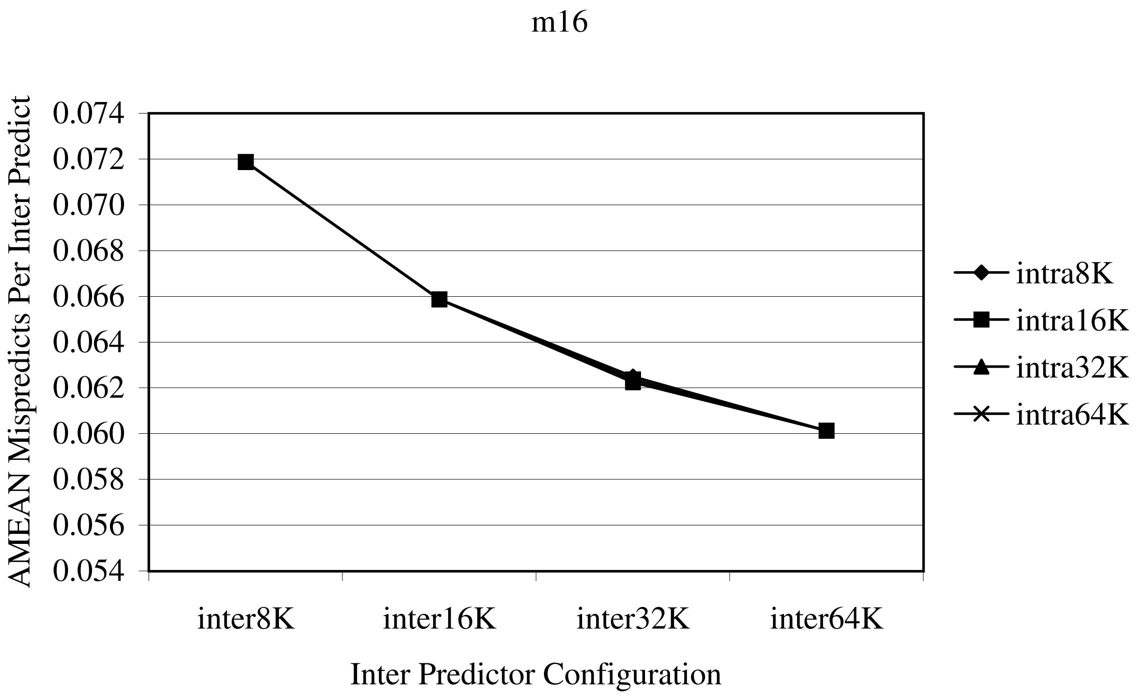


Figure 5.18: SPEC CINT95 inter-task misprediction ratio with 16 processing units for different predictor sizes.

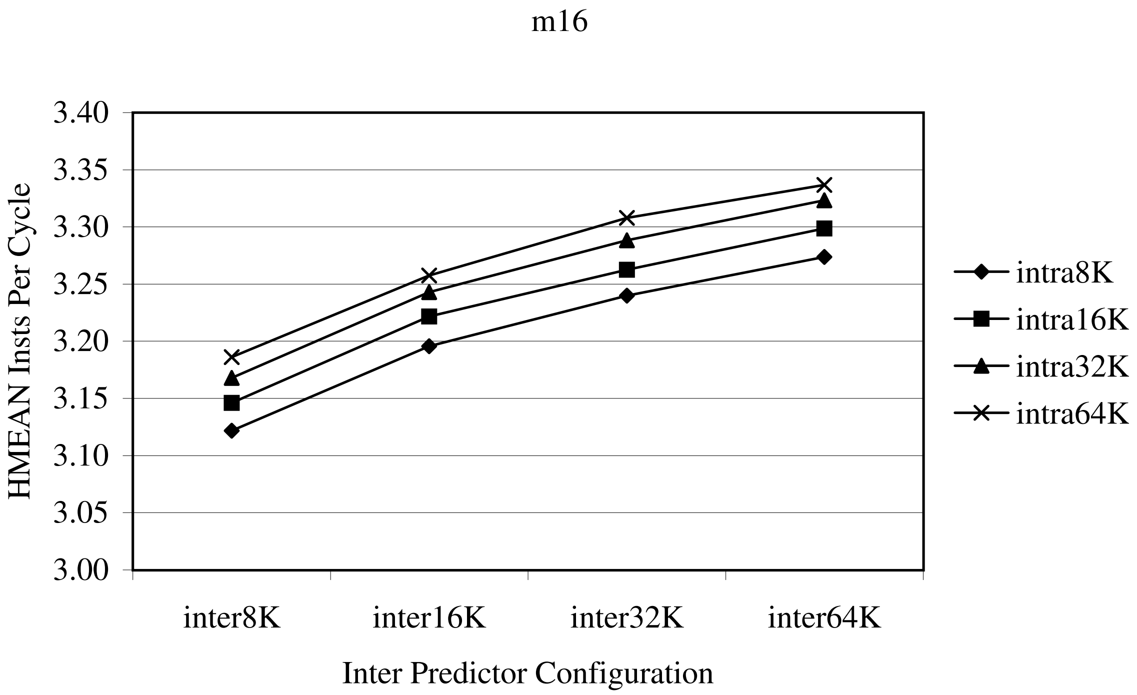


Figure 5.19: SPEC CINT95 performance with 16 processing units for different predictor sizes.

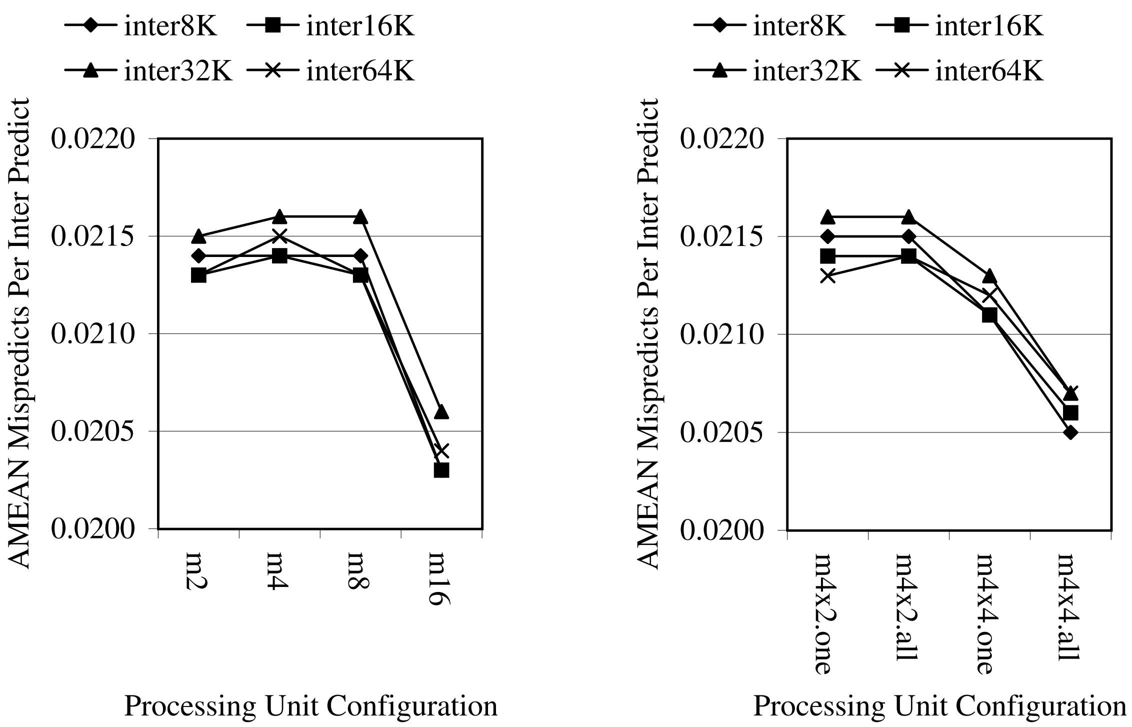


Figure 5.20: SPEC CFP95 inter-task misprediction ratio for different inter-task predictor sizes.

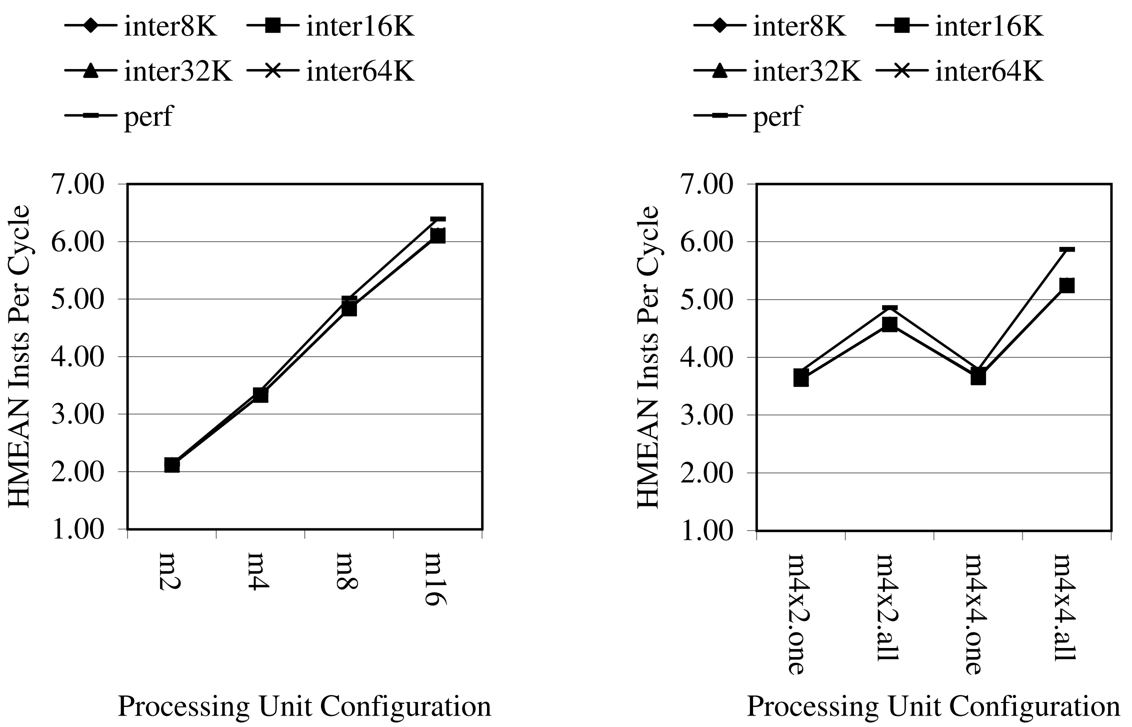


Figure 5.21: SPEC CFP95 performance for different inter-task predictor sizes.

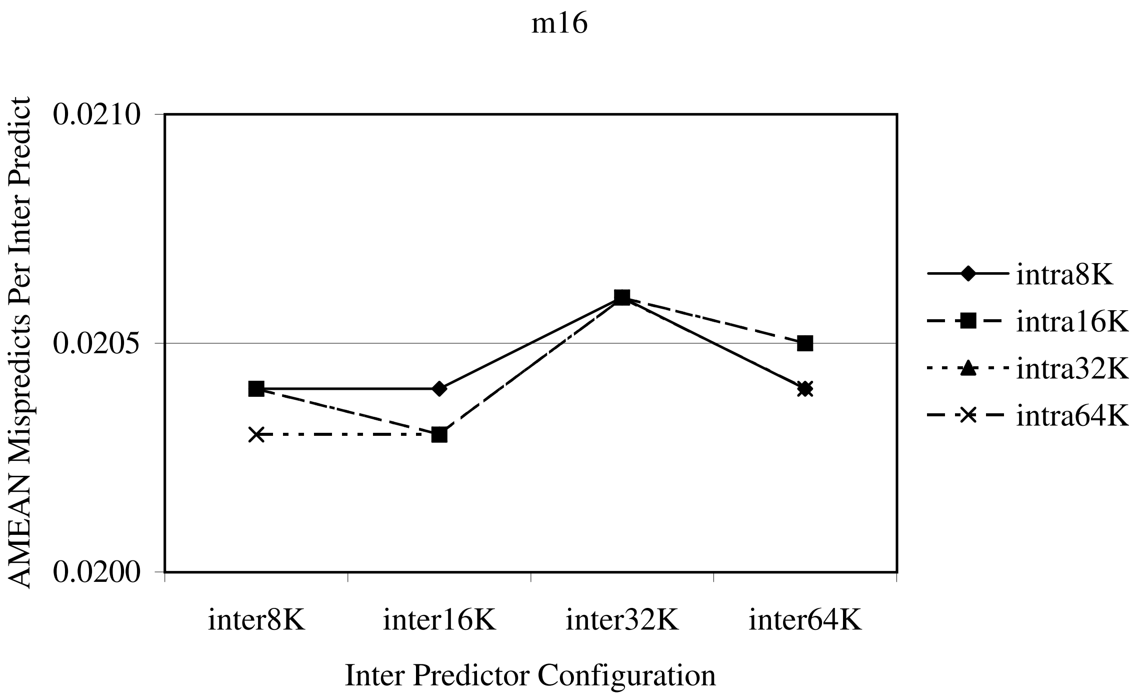


Figure 5.22: SPEC CFP95 inter-task misprediction ratio with 16 processing units for different predictor sizes.

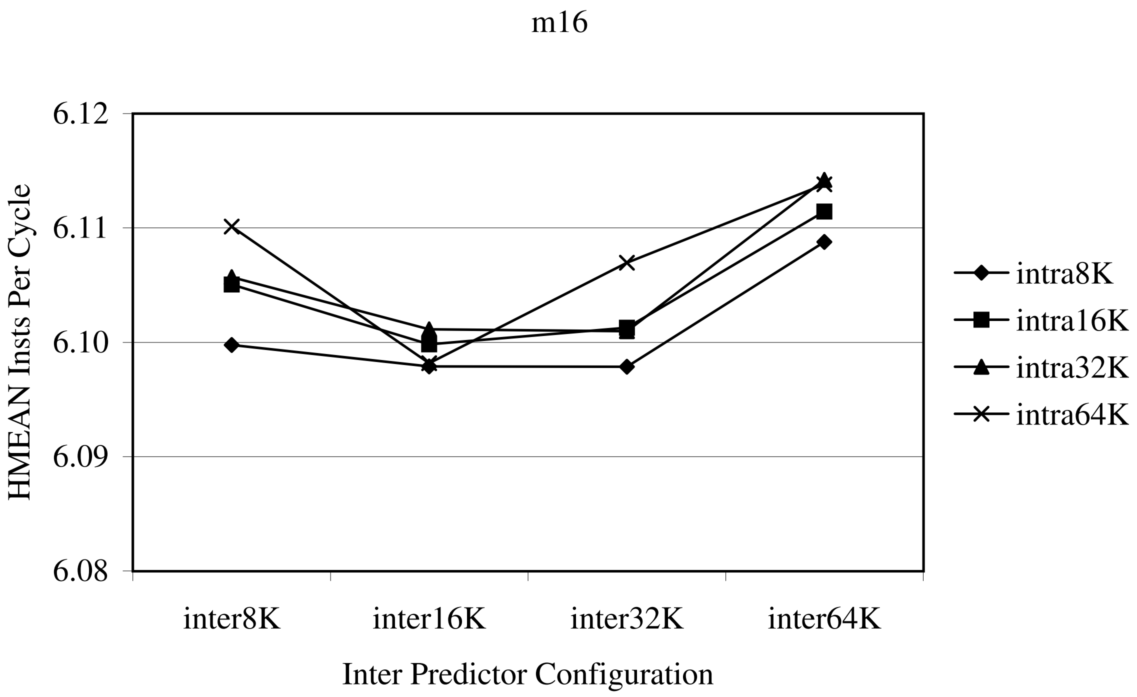


Figure 5.23: SPEC CFP95 performance with 16 processing units for different predictor sizes.

5.6.4.3 Intra-Task Results

The intra-task results are gathered in two steps (as for the inter-task results). First, to study different intra-task predictor sizes for a range of processing unit configurations, the results focus on designs where the inter-task prediction state is fixed to 64K-entry, while the intra-task prediction state is varied as 8K-entry, 16K-entry, 32K-entry, and 64K-entry. Second, once the trends for different processing configurations have been established, the results focus on one processing unit configuration, 16 processing units, to study different intra-task predictor sizes in conjunction with different inter-task predictor sizes.

Figures 5.24 and 5.25 give the misprediction ratio and the performance in instructions per cycle, respectively, with each of the different intra-task predictor sizes for the SPEC CINT95 benchmarks. The misprediction ratio increases steadily as the intra-task predictor size is halved, though the magnitude of this difference is not very significant. Comparing the smallest (8K-entry) to the largest (64K-entry) predictor, the difference is less than 0.025 mispredictions per prediction. The differences between the different predictor sizes remain constant even as the prediction accuracy changes for different processing unit configurations. Moreover, the overall performance shows no significant differences between the different predictor sizes, though the differences do increase somewhat as the number of processing units is increased (as described earlier).

Figures 5.26 and 5.27 focus on the 16 processing unit configuration for the SPEC CINT95 results just described. In the earlier results, the inter-task prediction state was fixed at 64K-entry. Now, the inter-task prediction state is varied among 8K-entry, 16K-entry, 32K-entry, and 64K-entry as well. The intra-task misprediction ratio is not influenced much by changes to the inter-task predictor size. However, the intra-task predictor is somewhat dependent on the inter-task predictor because of the global control information provided for each task (described earlier). As a result of this dependence, there is a slight trend towards better intra-task predictor accuracy for all predictor sizes as the inter-task predictor size is increased (presumably because a bigger inter-task predictor provides better global control information). Again, overall performance is dependent on the behavior of both predictors. As shown in the figure, there is a steady increase in performance as the inter-task predictor size is increased that is fairly constant as the intra-task predictor size is increased.

Figures 5.28 and 5.29 give the misprediction ratio and the performance in instructions per cycle, respectively, with each of the different intra-task predictor sizes for the SPEC CFP95 benchmarks. The misprediction ratio does not show significant differences as the intra-task predictor size is halved, though there is a consistent trend that the misprediction ratio increases as the intra-task predictor size decreases. Note that, for the scale of the graph, the magnitude of these differences is around 0.002 mispredictions per prediction comparing the smallest (8K-entry) to the largest (64K-entry) predictors. Looking at the overall performance, there is no distinguishable differences between the different predictor sizes.

Figures 5.30 and 5.31 focus on the 16 processing unit configuration for the SPEC CFP95 results just described. In the earlier results, the inter-task prediction state was fixed at 64K-entry. Now, the inter-task predictor size is varied among 8K-entry, 16K-entry, 32K-entry, and 64K-entry as well. The intra-task misprediction ratio is not influenced by the changes to the inter-task predictor size in any significant and/or consistent manner, even though the intra-task

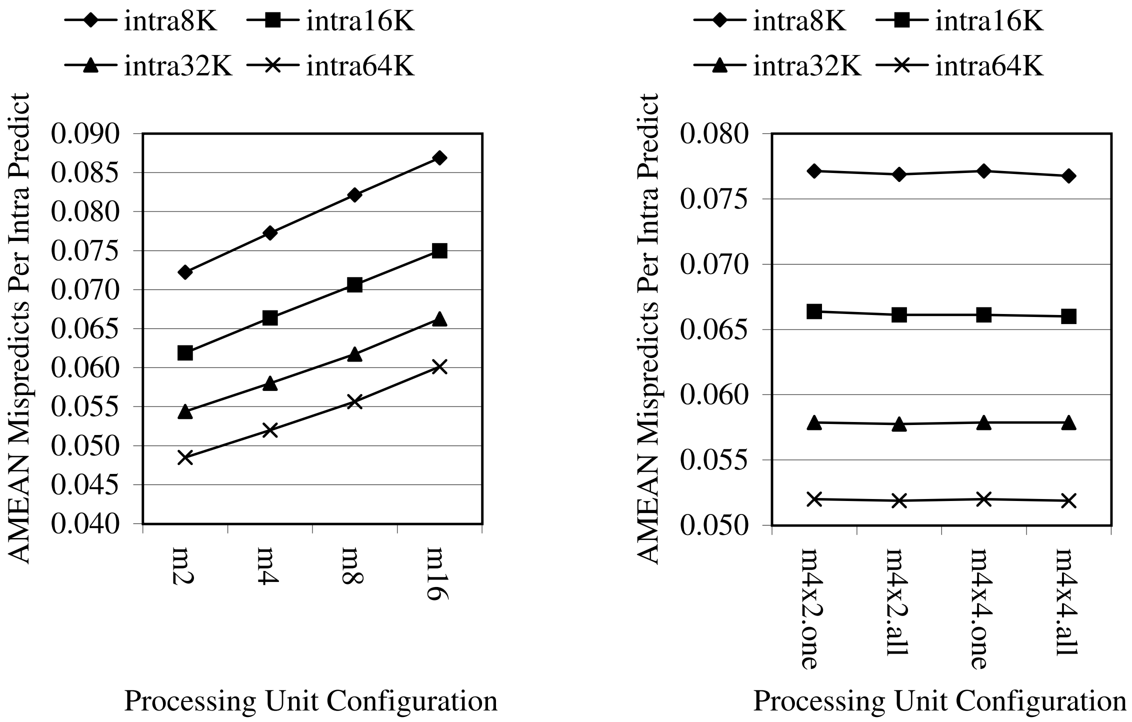


Figure 5.24: SPEC CINT95 intra-task misprediction ratio for different intra-task predictor sizes.

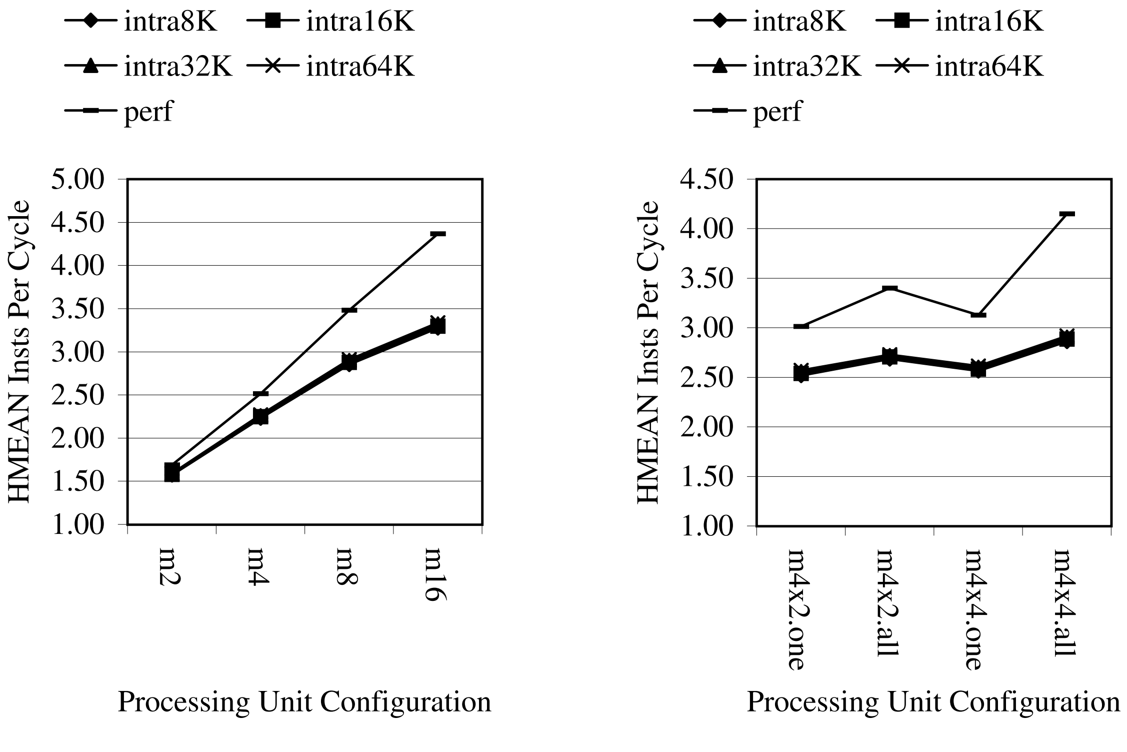


Figure 5.25: SPEC CINT95 performance for different intra-task predictor sizes.

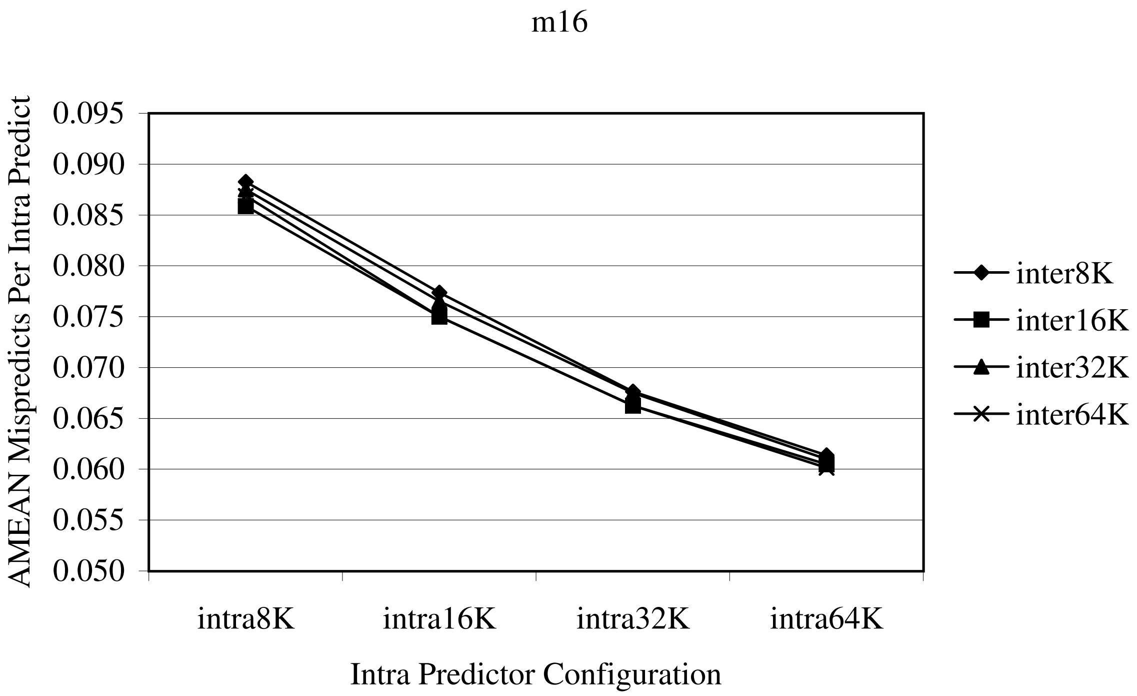


Figure 5.26: SPEC CINT95 intra-task misprediction ratio with 16 processing units for different predictor sizes.

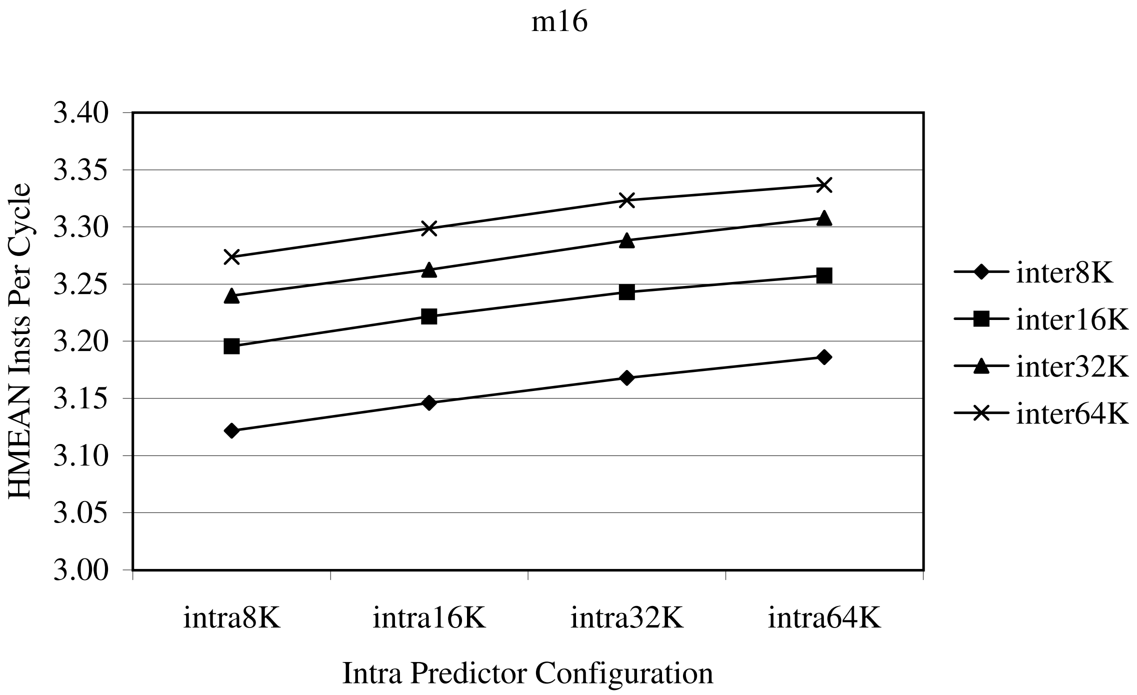


Figure 5.27: SPEC CINT95 performance with 16 processing units for different predictor sizes.

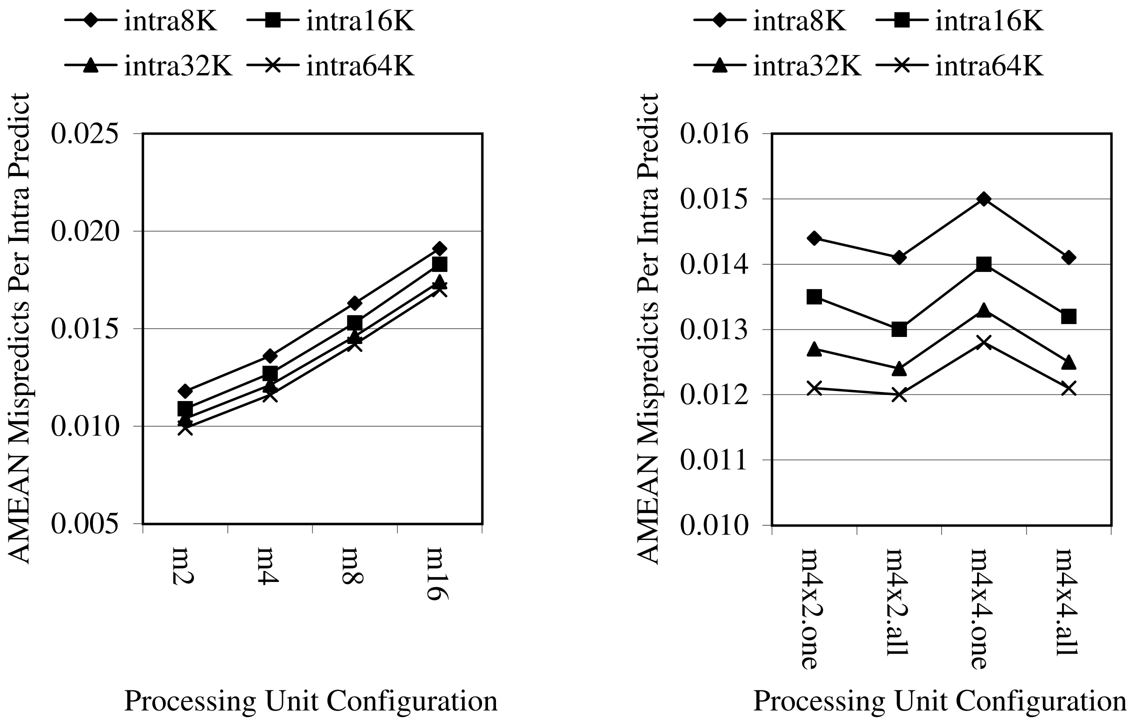


Figure 5.28: SPEC CFP95 intra-task misprediction ratio for different intra-task predictor sizes.

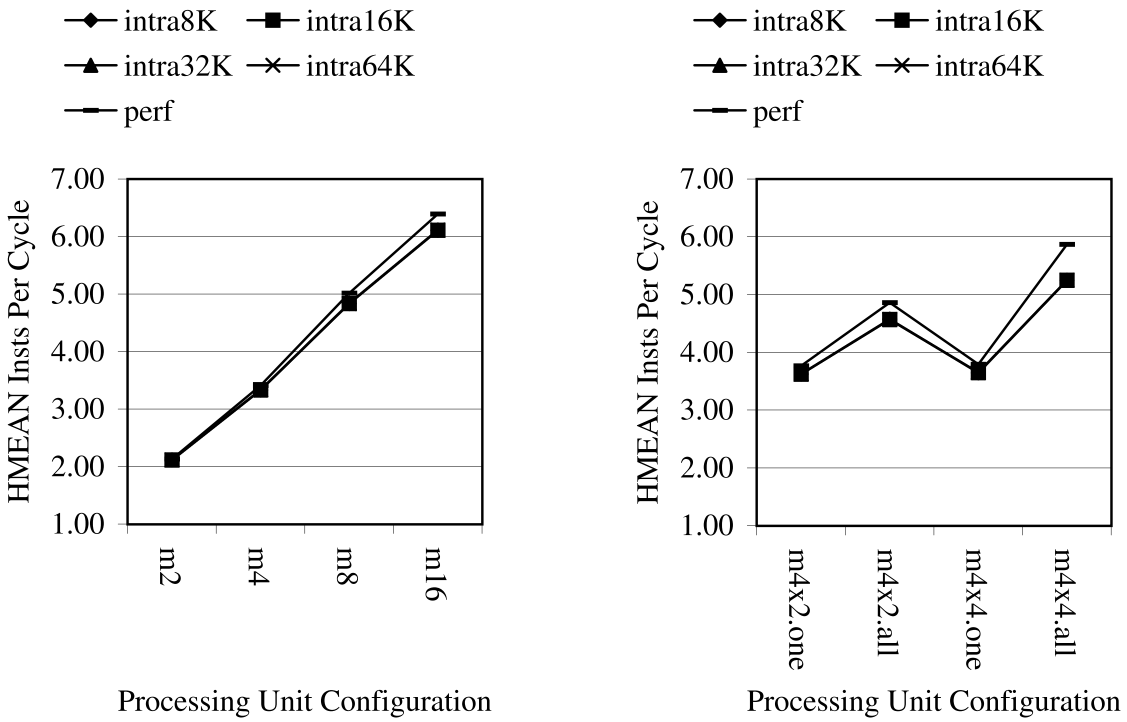


Figure 5.29: SPEC CFP95 performance for different intra-task predictor sizes.

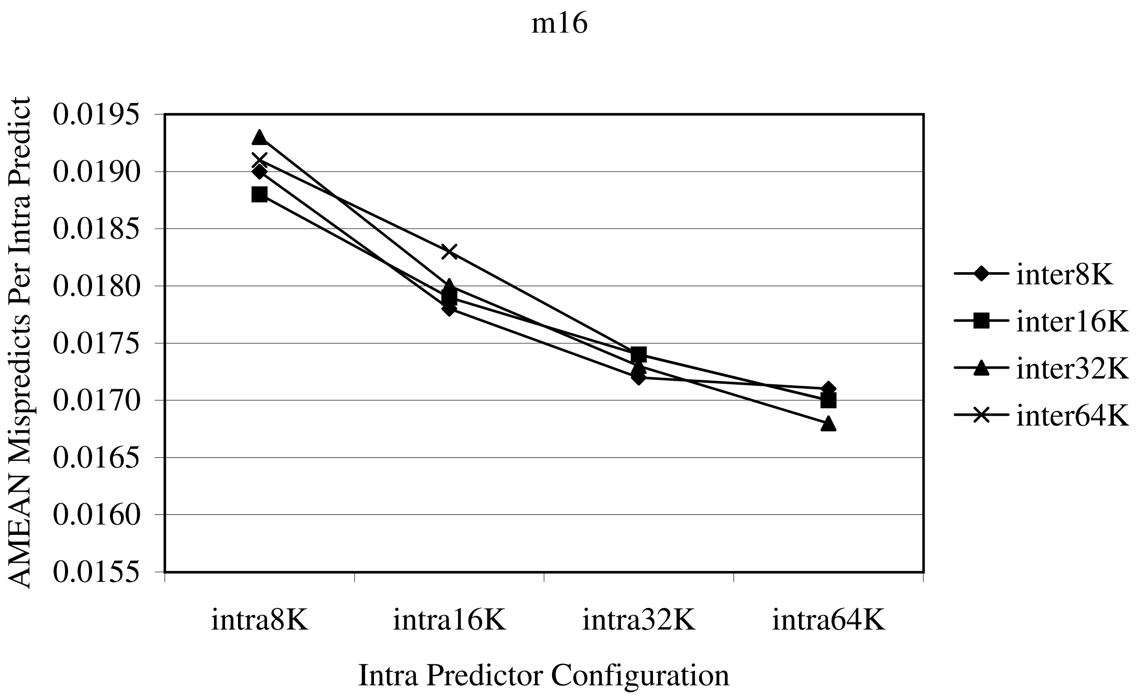


Figure 5.30: SPEC CFP95 intra-task misprediction ratio with 16 processing units for different predictor sizes.

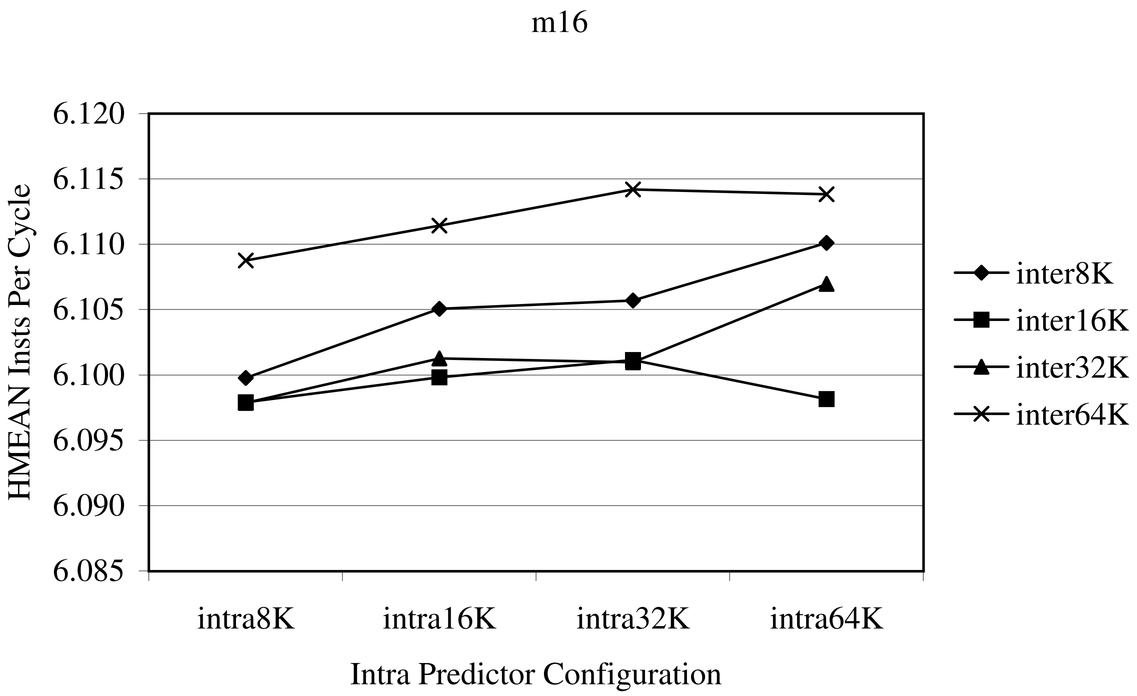


Figure 5.31: SPEC CFP95 performance with 16 processing units for different predictor sizes.

predictor has some dependence on the global control information (described earlier) provided by the inter-task predictor. The reason for this behavior is that the intra-task predictor is hardly influenced much by changes to its own size let alone changes to the inter-task predictor size. Note again that, for the scale of the graph, the magnitude of these differences is less than 0.0002 mispredictions per prediction comparing the smallest (8K-entry) to the largest (64K-entry) intra-task predictors. The overall performance differs by less than 0.01 instructions per cycle or a negligible percentage among intra-task predictor sizes for each inter-task predictor size.

5.7 Instruction Memory Design

The instruction memory must support two distinct types of access, inter-task and intra-task, in accordance with the way addresses are provided by the hierarchical predictor. As discussed earlier, the inter-task accesses are serviced by the task cache, while the intra-task accesses are serviced by the instruction caches. The reference characteristics of the two are somewhat different from those of caches used in conventional processors, but nonetheless may be accommodated in a straightforward manner (except perhaps for the scale) with adaptations of existing techniques.

5.7.1 Cache Memory

A cache is a small, fast buffer in which a system tries to keep those parts of the contents of a larger, slower memory that will be used soon. The purpose of a cache is to improve system cost/performance by providing the capacity of the large, slow memory with close to the access time of the small, fast cache. An overview of caching can be found in [78]. Caches are used in nearly all microprocessor implementations today because of their ability to decrease effective access time as well as to reduce traffic to other levels of the memory hierarchy.

Much work has focused on miss rates as related to overall size, associativity, block size, indexing function, and replacement policy (to name the most well-studied aspects) for caches that provide bandwidth of only a single access per cycle. This type of work is certainly relevant to the instruction memory design, for both the task cache and the instruction caches, though it is necessary to consider additionally the effects of providing bandwidth of multiple accesses per cycle for the instruction caches. Moreover, the demands placed on the instruction memory by a multiscalar processor require further attention to the access latency for the both the task cache and the instruction caches.

The task cache is relatively straightforward because it only services at most a single access per cycle. Therefore, the access latency can simply be varied to see the effect on overall performance. However, the instruction caches are more involved because of their need to service multiple accesses per cycle, (at least) one from each processing unit. The use of a high-bandwidth cache organization can provide multiple accesses per cycle, but there are trade-offs in terms of miss rate and access latency depending on the design. (This study does not go into the detail to consider the relationships between access latency and cache size, associativity, *etc*.)

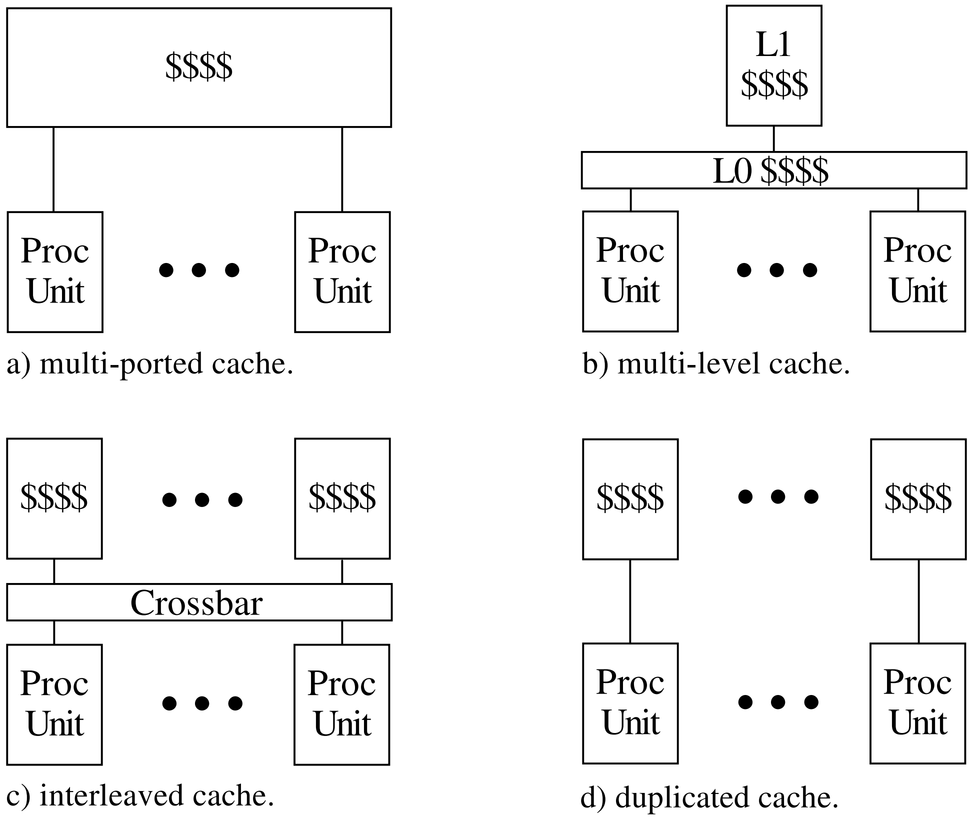


Figure 5.32: High-bandwidth cache organizations.

5.7.2 High-Bandwidth Cache Organizations

High-bandwidth cache organizations are well-developed in the literature and in practice. The common approaches, illustrated in Figure 5.32, are multi-ported, multi-level, interleaved, and duplicated [88]. According to the order listed, these organizations provide a progression from more centralized, with higher access latency and lower miss rates, to more decentralized, with lower access latency and higher miss rates.

Though both multi-ported and multi-level designs are described below, these approaches are not well suited for use in a multiscalar processor since neither scales well to many ports for the kinds of cache sizes studied here. (The multiscalar processor designs considered might need to scale up to perhaps 16 ports for 16 processing units with total cache sizes up to 128K-byte.) The interleaved and duplicated designs, referred to as shared and private respectively here, fit within the design constraints of the instruction memory.

5.7.2.1 Multi-Ported

A multi-ported cache organization provides high-bandwidth access using a brute force approach, as shown in part a) of Figure 5.32. Each processing unit is provided its own data path to every entry in the cache, implemented by either replicating the entire cache structure (one single-ported cache for each port) or multi-porting the individual cache cells. Since every entry of the cache is accessible from each port of the device, this design provides bandwidth in a uniform fashion.

The capacitance and resistance load on each access port increases as the number of ports or entries is increased, resulting in longer access latency [99]. In addition, this design has a large area due to the many wires and comparators needed to implement each port. The area of a multi-ported device is usually proportional to the square of the number of ports [39].

This organization ought to provide ideal bandwidth and access characteristics for a given cache size. However, the drawbacks with respect to its access latency and implementation area considerations, make it an unsuitable choice for the multiscalar processors consider in this study. Yet, it is sometimes used as a standard by which to gauge the performance of other organizations.

5.7.2.2 Multi-Level

A multi-level cache organization provides high-bandwidth and low-latency access by exploiting locality in program references, as shown in part b) of Figure 5.32. When an entry from a base cache (L1 cache) is referenced, it is placed into a small upper-level cache (L0 cache). If the L0 cache offers a good (enough) miss rate, then it may reduce the bandwidth demand on the L1 cache enough such that it can be designed with only one port [41].

When an access misses in the L0 cache, it must forward the request to the L1 cache, where L1 cache access port contention, L1 cache access latency, and L1 cache miss latency may increase the latency of the access. Since the L0 cache is small, it may be possible to use a highly associative design to improve the hit rate of the L0 while keeping the L1 associativity constant.

The L0 cache is a multi-ported cache with enough ports to handle all simultaneous requests from the processor. The additional area overhead of this organization is concentrated in the implementation of the L0 cache, which for small sizes and few ports should be much smaller than the L1 cache, but for large size and many ports degenerates to a multi-ported cache. A variant of this approach, a hybrid between the next two organizations described, is considered later.

5.7.2.3 Interleaved

An interleaved cache employs an interconnect to interleave the storage among multiple shared cache banks, as shown in part c) of Figure 5.32. Each cache bank can independently service some number (often one) of requests per cycle. This organization provides high-bandwidth access as long as simultaneous accesses map to different banks. When accesses map to the same bank, a so-called bank conflict, stalls may occur. The mapping between addresses and the cache banks is defined by the bank selection function. This function influences the distribution of the accesses to the banks, and hence, the bandwidth delivered by the device. A common function is to interleave cache blocks based on the low-order bits of the accessed cache block address; various other functions have been proposed (such as those described in [72]), but are not considered further here.

This organization ought to have better latency and area characteristics than a multi-ported organization, especially for large caches [100]. While the interconnect, usually a full crossbar, adds some latency to the access path, this latency may be mitigated by the shorter access latency of the smaller, single-ported banks. The area overhead is concentrated in the interconnect; for a full crossbar, the implementation area is usually proportional to the square of the number of access ports. In an actual design, there may be different trade-offs in terms of size and access time depending on the number of ports and the implementation technology.

5.7.2.4 Duplicated

A duplicated cache provides high-bandwidth access by associating a private cache with each processing unit, as shown in part d) of Figure 5.32. Each private cache can independently service requests. In contrast to the interleaved approach, there are no bank conflicts or interconnect problems. However, because each cache is a private copy, it is unclear to what extent spatial and/or temporal locality may be exploited from the reference stream. That is, (i) accesses of a cache block may miss in each of the private caches, and (ii) replacements of a cache block may occur before a profitable reuse.

It may be possible to compensate for this behavior with the right choices in associativity, cache block size, and overall cache size. Nevertheless, the need to support multiple copies of a cache block in each of the private caches likely means poor overall utilization of the cache storage available as well as possible concerns about keeping these copies consistent (though not for read-only accesses such as those of the instruction caches). Despite these problems, a duplicated cache organization ought to provide the lowest access latency among the organizations described.

5.7.3 Multiscalar Considerations

The caches used for the instruction memory are really not much different from the designs of conventional processors. As a result, there are no major modifications of the cache designs needed to support a multiscalar processor. Nevertheless, there are a few considerations worth pointing out with respect to the task cache and instruction caches studied here in the context of multiscalar processors.

5.7.3.1 Task Cache

The task cache provides expeditious access to task summary information generated by the compiler. This cache supports only a single access per cycle (as part of inter-task sequencing), but must deliver high-bandwidth in terms of the number of words supplied by each access, since a task descriptor may be quite large. Recall the task descriptor contains both control and data information about the task. A task cache block is equal in size to the task descriptor (described in an earlier chapter), which for this implementation is 64 bytes. Moreover, each task cache access provides the entire task cache block.

The reason for this behavior is that both the control part and the data part of the task descriptor must be available before the task assignment may continue. In order for inter-task sequencing to take the next step through the control flow graph, it needs the summary information about its most recent step. This dependence is a by-product of the design choice to allow the compiler (rather than the hardware) to orchestrate control flow and data flow among the tasks. Because the hardware is designed to rely on the compiler to provide addresses for many of the task exits and to coordinate register communication among tasks, it cannot continue to assign tasks without this information and must stall in such cases.

5.7.3.2 Instruction Cache

The instruction caches provide expeditious access to the actual instructions of the program. These caches support multiple accesses per cycle (at least one per cycle from each processing unit as part of intra-task sequencing), but need only deliver a few words for each access, dividing the total instruction cache bandwidth among each of the processing units.

In contrast to the task cache, the instruction caches can tolerate misses. The steps through the control flow graph taken by each processing unit for intra-task sequencing are more or less independent of steps taken by other processing units. As a result, only the processing unit whose intra-task sequencing misses in the instruction caches need wait while the miss is serviced.

In order to streamline the high-bandwidth organizations for use in the multiscalar instruction supply, the instruction caches are modified to combine hit and miss processing whenever possible using techniques that have been proposed for multiprocessors. In particular, the shared cache organizations combine cache accesses on both hits and misses to the same cache block [47]; the private organizations combine accesses (snarf) on misses to the same cache block [29].

5.8 Instruction Memory Evaluation

The instruction memory evaluation is divided into two parts, thereby focusing on the task cache and the instruction cache (or inter-task and intra-task) aspects of the instruction memory separately. Though there is a connection between the task cache and the instruction cache from the standpoint of exposing the dynamic instruction window, the cache behaviors are independent of one another. Thus, the approach does reflect the expected characteristics of the caches when combined in an actual design.

First, the task cache study is performed to focus on the extent to which the task cache affects the execution of the processor. For this study, the evaluation involves varying the task cache size and associativity, for a constant block size and replacement policy. (The task cache block size may not actually be varied since a block corresponds to a single task descriptor.) Moreover, the access latency of the task cache is changed to investigate the effect that this characteristic has on the execution of the processor.

Second, the instruction cache study is performed to focus on the extent to which the instruction cache, for both shared and private organizations as well as a hybrid organization, affects the execution of the processor. For this study, the evaluation involves varying the instruction cache size and associativity, for a constant block size and replacement policy. Moreover, the access latency of the shared instruction cache is changed to investigate the effect that this characteristic has on the execution of the processor as compared to the private instruction cache design with constant latency.

5.8.1 Metrics

The most common metric used for cache evaluation is the miss ratio of the cache – the number of misses per access performed on the cache. In this evaluation, there are actually two miss ratios considered, the architectural miss ratio and the speculative miss ratio. The architectural miss ratio divides the total number of misses (useful and non-useful) by the number of accesses performed with respect to useful tasks or useful instructions. The speculative miss ratio divides the total number of misses (useful and non-useful) by the total number of accesses regardless of whether there is a connection to useful tasks or useful instructions. The reason both metrics are used is to account for the effect of non-useful accesses on the miss rate observed by useful accesses. The relative behavior for the two metrics is the same because only the divisor changes. Nevertheless, the difference in absolute terms between the two metrics can help to clarify whether the cache is servicing useful and not just non-useful accesses in an effective manner.

This evaluation provides the miss ratios as the unweighted arithmetic mean (AMEAN) for each of the two groups of benchmarks, SPEC CINT95 and SPEC CFP95, respectively. The arithmetic mean is the correct mean (as opposed to the harmonic or geometric) because the miss ratio is proportional to the program execution time; the mean is unweighted because each benchmark program is meant to be representative of a particular type of (integer or floating point) application independent of the number of instructions it executes for its particular input. Despite the fact that this evaluation uses the miss ratio for evaluation, the miss ratio may be a misleading indicator of a cache's impact on processor performance. Two different cache designs may have the same miss ratio but provide different performance if the times to service a miss are different. Moreover, miss ratio does not take into account the contention that may occur in the presence of aggressive speculative execution.

A more useful metric to compare cache organizations that takes into account the demands placed upon the cache by the overall processor organization is the effective access latency. The effective access latency is the amount of time (cycles in this evaluation) between when an access is made and the cache returns the information requested in the access. For this evaluation, the time is measured over all accesses, whether useful or non-useful, to provide the arithmetic means of these times. As with miss ratio, this metric is provided for the two groups of benchmarks using the unweighted arithmetic mean (AMEAN), since effective access latency, like miss ratio, is proportional to the program execution time. Though the effective access latency is a more telling metric for evaluation than miss ratio, it still does not provide a direct measure of the cache organization's impact on processor performance. Thus, the processor performance given as the harmonic mean (HMEAN) of instructions per cycle (already described in Section 4.6.1) is used as well to differentiate between the utility of different cache designs.

5.8.2 Task Cache Study

Since this part of the instruction memory evaluation focuses only on the task cache, the other processor components, including the instruction cache, are fixed to the standard configuration. The details of the exact configurations are given in Table 3.11 of Chapter 3. The task cache configurations, being the subject of this study, are varied as described below.

5.8.2.1 Configurations

The task cache is configured with different sizes, associativities, and access times. The block size (a block is a task descriptor in this case) and the replacement policy (least recently used) are not varied in this evaluation. In this evaluation, the cache size is varied among 64, 128, 256, 512, and 1024 entries; the associativity is varied among 1-way, 2-way, and 4-way designs. Moreover, for each of these organizations, a range processing unit configurations are used. In particular, the number of processing units is varied among 2, 4, 8, and 16, as well as among 4 processing units running 2 and 4 tasks one-at-a-time and all-at-once. For all of the studies performed, the base task cache access latency is 1 cycle (on a hit), though it is varied among 2, 3, and 4 cycles (pipelined where the pipeline is flushed on a miss) to study this effect on overall processor performance. Using these configurations, the task cache results are gathered in two steps.

First, the task cache evaluation focuses in general on the impact of different sizes among the processing unit configurations. For this evaluation, only 2-way associative task caches are used (the results for the other associativities are similar, though shifted higher or lower as seen in the second part of the evaluation). Only processing unit configurations that run a single task per processing unit are discussed. The overall differences among configurations are not that significant, and the multiple tasks per processing unit are similar to the results for the single task per processing unit configurations. In addition, this part of the task cache evaluation focuses on the behavior of one task cache configuration, 1024 entries with 2-way associativity, for different hit latencies. Second, the task cache evaluation focuses in specific detail on the behavior with different sizes and associativities for only one processing unit configuration, 16 processing units. The relative differences for all metrics with changing cache sizes and associativities follow the same trends for all of the processing unit configurations.

5.8.2.2 General Results

Figure 5.33 gives the speculative (on the left) and architectural (on the right) miss ratios with different task cache sizes and 2-way associativity for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Figure 5.34 gives the same information for the SPEC CFP95 benchmarks. The speculative miss ratio, misses per access, varies significantly between the smallest (64 entries) and largest (1024 entries) task cache sizes, from almost no misses per access for integer and floating point benchmarks to 0.20 misses per access for

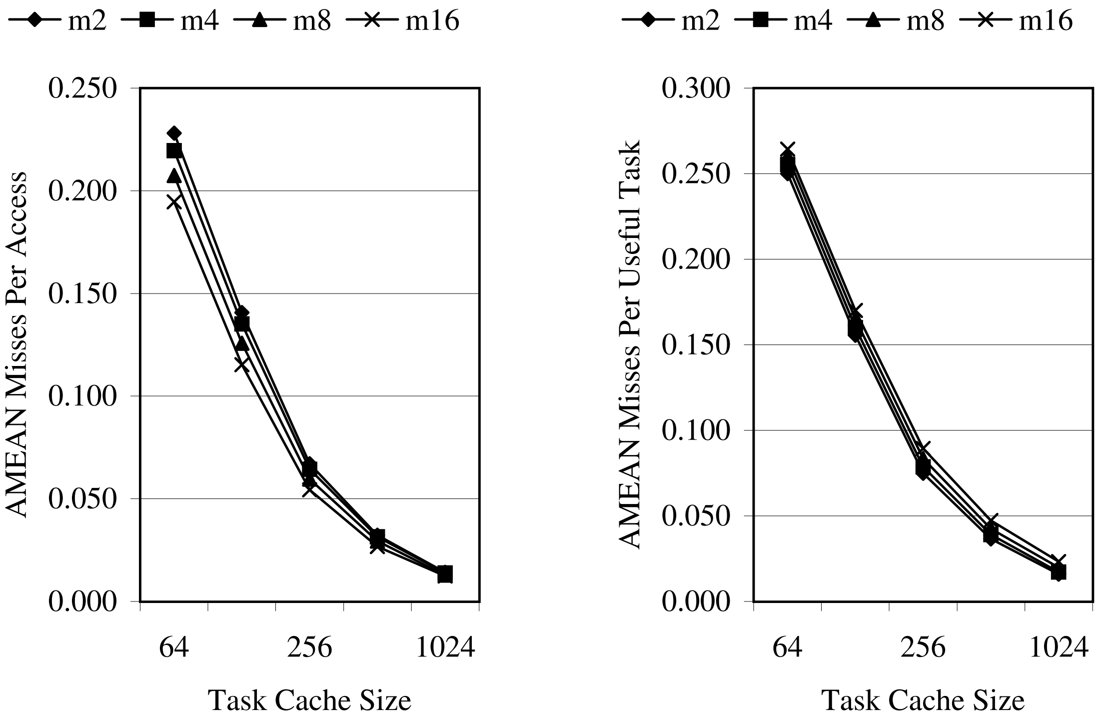


Figure 5.33: SPEC CINT95 misses per access and misses per useful task with 2-way associative task caches for 2, 4, 8, and 16 processing units.

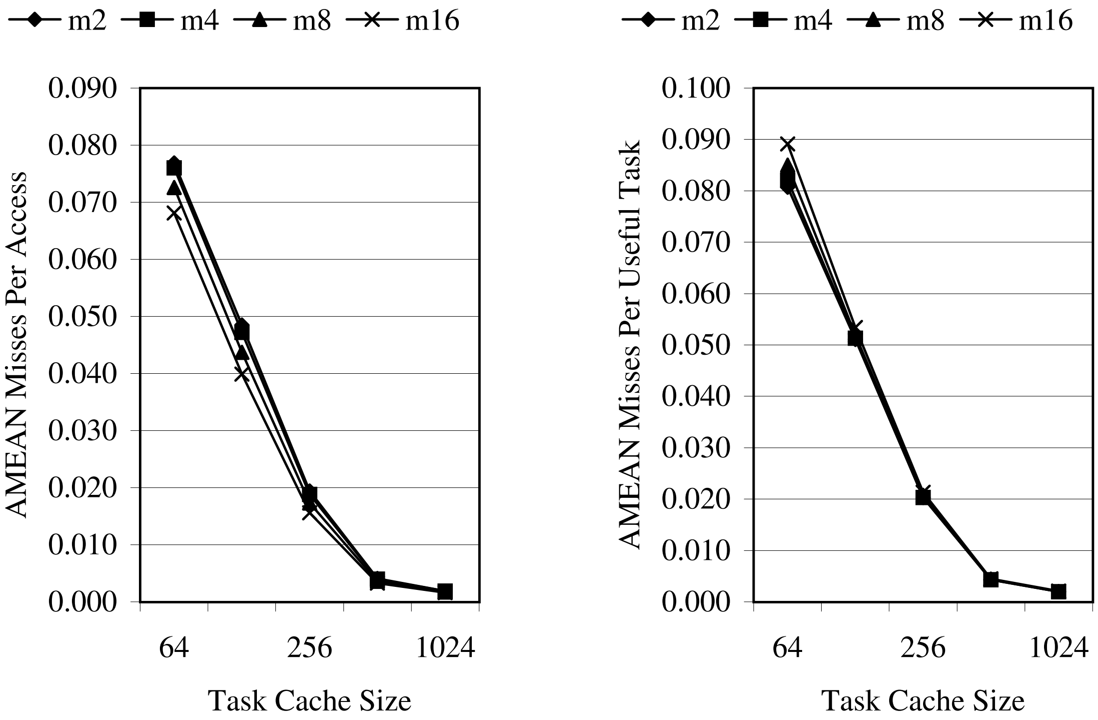


Figure 5.34: SPEC CFP95 misses per access and misses per useful task with 2-way associative task caches for 2, 4, 8, and 16 processing units.

integer benchmarks and 0.08 misses per access for floating point benchmarks. The architectural miss ratio, misses per useful task, varies in the same manner, though the miss ratios are higher, with more of a relative increase for integer than for floating point benchmarks.

Overall, the SPEC CINT95 and SPEC CFP95 benchmarks exhibit similar behavior for both speculative and architectural miss ratios, but the variations across different numbers of processing units is much smaller for floating point as compared to integer results. Nevertheless, for the speculative miss ratios, there is a trend that the miss ratios decrease as the number of processing units increases; this trend may be seen by observing that lines one the graph are ordered top (higher speculative miss ratio) to bottom (lower speculative miss ratio) as m2, m4, m8, and m16. In contrast, for the architectural miss ratios, there is a trend that the miss ratios decrease as the number of processing units decreases; in this case, the lines are ordered m16, m8, m4, and m2, top to bottom. Comparing the nature of the speculative and the architectural miss ratios gives some insight into this phenomenon.

In general, different numbers of processing units are capable of performing different amounts of speculation. For instance, the 16 processing unit configuration performs more speculative accesses than the 2 processing unit configuration because it speculates the execution of many more tasks. Many of the speculative accesses for the 16 processing units do not miss in the task cache, so the speculative miss ratio is lower than for the 2 processing unit configuration. Nevertheless, the 16 processing unit configuration also generates more misses overall per useful task, so the architectural miss ratio is lower for the 2 processing unit configuration. It is difficult to ascertain in general whether such behavior helps or hurts performance, though it is difficult to avoid in any case as it is a natural by-product of speculative execution.

Figure 5.35 gives the effective access latency (on the left) and performance (on the right) with different task cache sizes and 2-way associativity for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Just beneath, Figure 5.36 gives the performance of a fixed 1024 entries and 2-way associativity task cache for hit latencies of 1, 2, 3, and 4 cycles (pipelined where the pipeline is flushed on a miss) on the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. In addition to these results for the integer benchmarks, the same results are given for the floating point benchmarks. Figure 5.37 gives the effective access latency (on the left) and performance (on the right) with different task cache sizes and 2-way associativity for the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Just beneath, Figure 5.38 gives the performance of a fixed 1024 entries and 2-way associativity task cache for hit latencies of 1, 2, 3, and 4 cycles (pipelined where the pipeline is flushed on a miss) on the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Though the absolute numbers for the two groups of benchmarks are obviously quantitatively different, it is interesting to compare them qualitatively.

The integer benchmarks are much more sensitive to the behavior of the task cache than are the floating point benchmarks. Much of this sensitivity is related to the difference in average task size for integer programs (about 12 instructions) and floating point programs (about 68 instructions). Because integer tasks are smaller than floating point tasks, the number of tasks descriptors that must be cached for the same number of instructions is larger (by a factor of

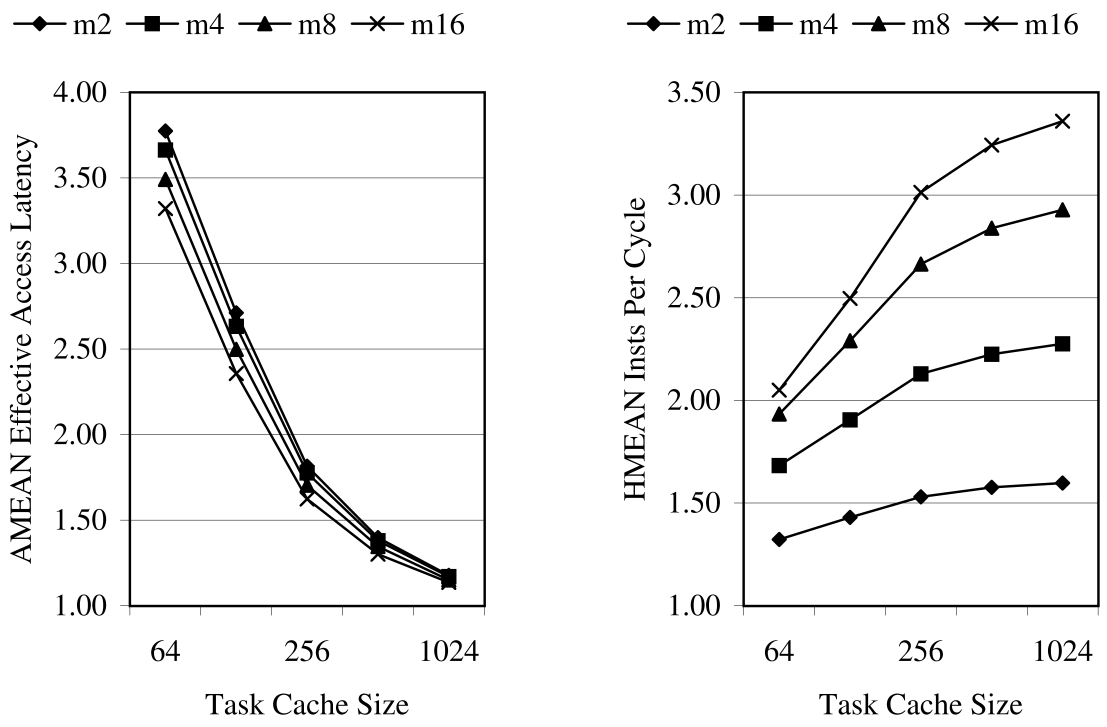


Figure 5.35: SPEC CINT95 effective access latency and performance with 2-way associative task caches for 2, 4, 8, and 16 processing units.

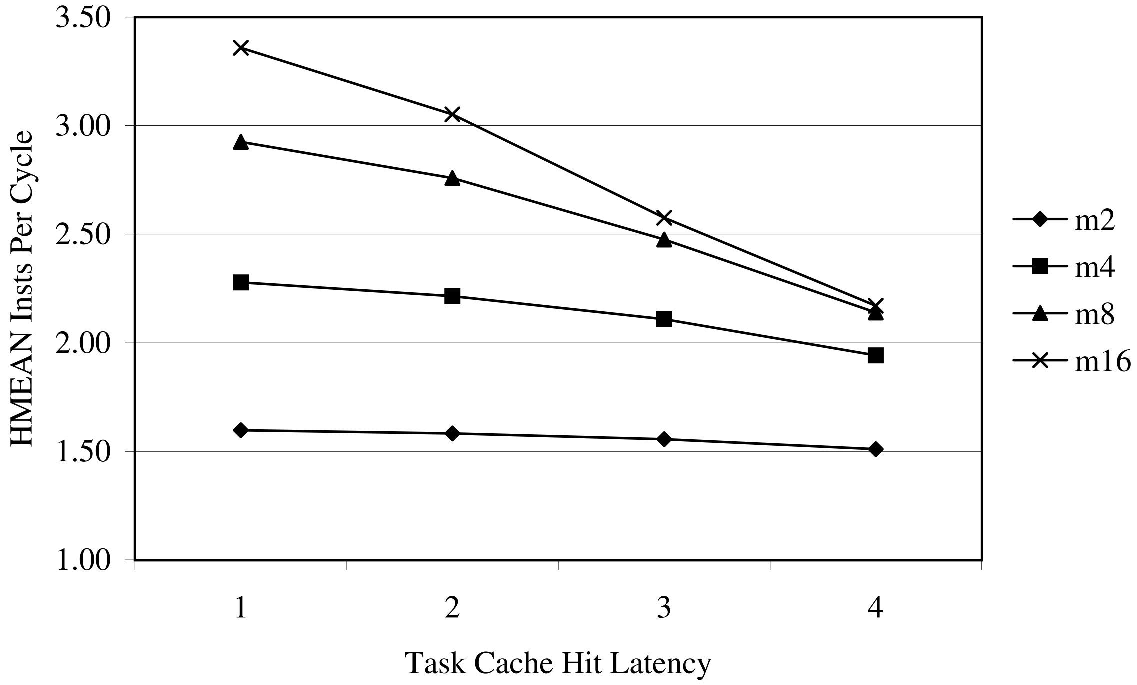


Figure 5.36: SPEC CINT95 performance with 1024 entry 2-way associative task cache for hit latencies of 1, 2, 3, and 4 cycles (pipelined where the pipeline is flushed on a miss).

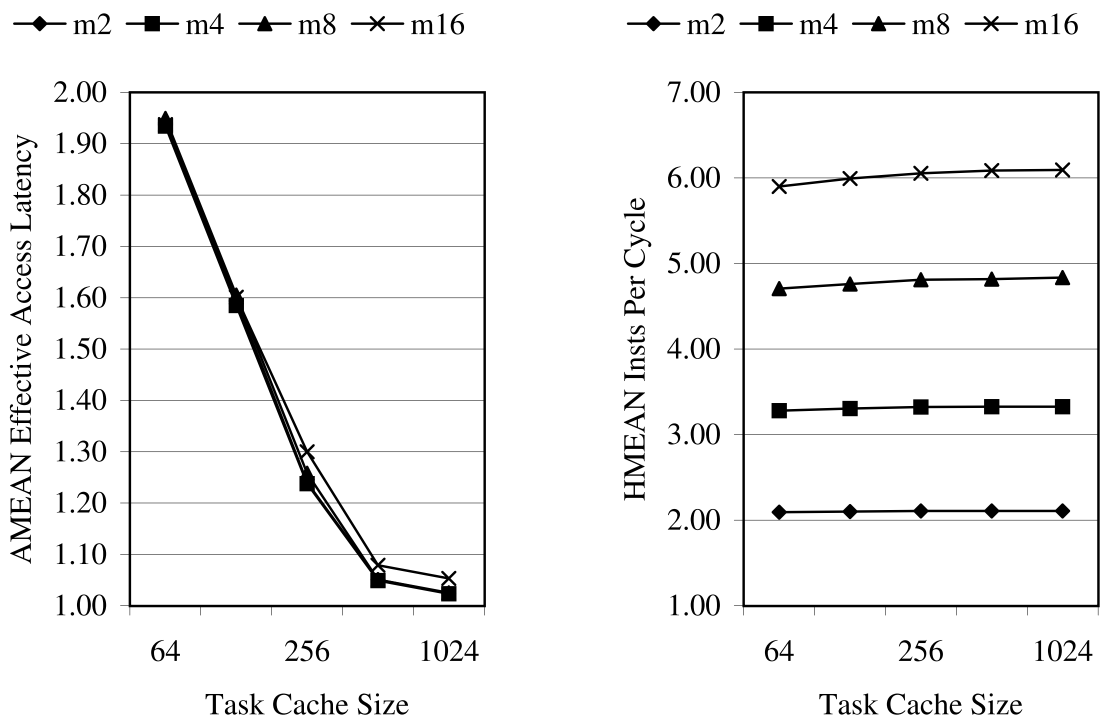


Figure 5.37: SPEC CFP95 effective access latency and performance with 2-way associative task caches for 2, 4, 8, and 16 processing units.

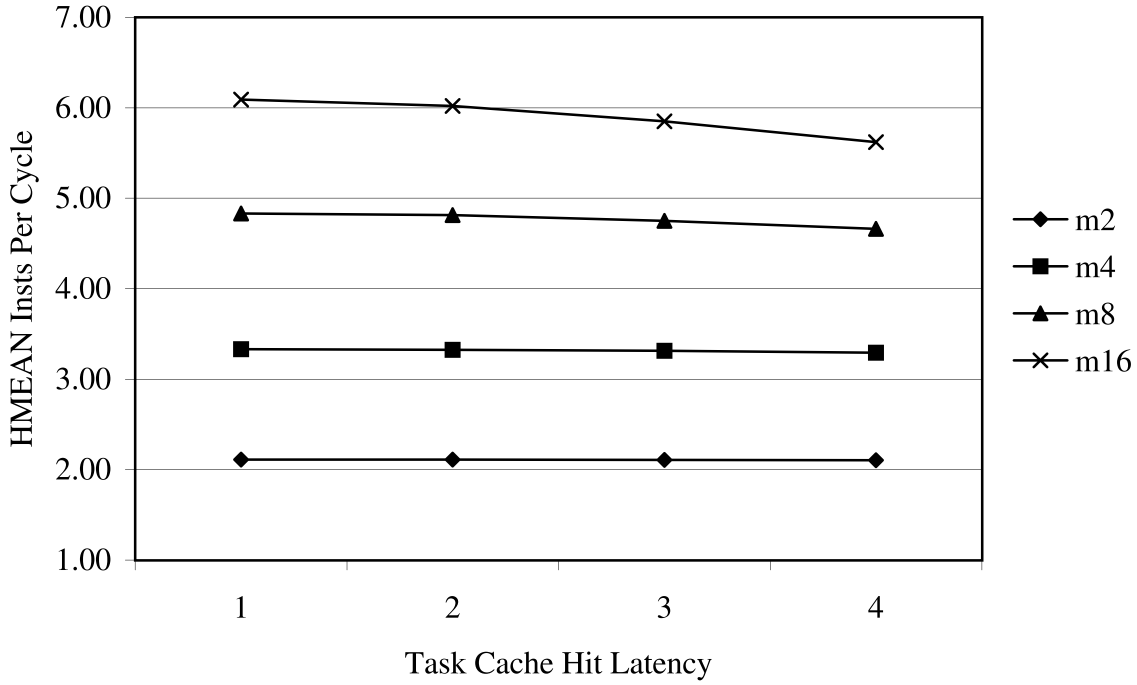


Figure 5.38: SPEC CFP95 performance with 1024 entry 2-way associative task cache for hit latencies of 1, 2, 3, and 4 cycles (pipelined where the pipeline is flushed on a miss).

around 5 times). This behavior is indicated by the miss ratios described above. As a result, the difference in miss ratios causes the effective access latency of the task cache to be much higher (by a factor of around 2 times) for integer as compared to floating point benchmarks. Moreover, the floating point benchmarks exhibit higher instruction-level parallelism (by a factor of about 2 times) than the integer benchmarks, so there is less parallel work around to keep the processing units busy when the task cache delays the start of a task. Considering the performance as the effective access latency increases, either due to an increase in access time to service misses or due to an increase in access time to service hits, the performance for the integer benchmarks falls off drastically, while the performance for the floating point benchmarks only falls off slightly. In fact, if the effective task access latency becomes too high, the benefit of providing more processing units simply becomes too low to justify the cost.

5.8.2.3 Specific Results

Figure 5.39 gives the speculative (on the left) and architectural (on the right) miss ratios with different task cache sizes and associativities for the SPEC CINT95 benchmarks using the 16 processing unit configuration. The speculative miss ratio, misses per access, decreases steadily as the task cache size is doubled for all associativities. Among the different associativities, there is a more significant difference going from 1-way to 2-way associativity (sometimes as much as 0.05 misses per access) and a less significant difference going from 2-way to 4-way associativity (usually around 0.01 misses per access). The associativity becomes less of a factor as the task cache size increases, but higher associativity consistently gives lower speculative miss ratios. Overall, the miss ratio for the smallest (64 entries) and the largest (1024) task caches differ by around 0.20 misses per access, a significant difference with respect to the overall miss ratio. The relative behavior for the architectural ratio, misses per useful task, is the same as for the speculative ratio (as to be expected). What is important to observe is that the misses per useful task is usually significantly higher than the number of misses per access. Thus, a significant fraction of the misses supported by the task cache are not a result of accesses initiated for the execution of a useful task.

Figure 5.40 gives the effective access latency (on the left) and the performance (on the right) with different task cache sizes and associativities for the SPEC CINT95 benchmarks using the 16 processing unit configuration. In accordance with the behavior of the miss ratios, the effective access latency decreases steadily as the task cache size is doubled for all associativities. Among the different associativities, there is a more significant difference going from 1-way to 2-way associativity (not more than 1.0 cycles per access) and a much less significant difference going from 2-way to 4-way associativity (around 0.2 cycles per access). The associativity becomes less of a influence as the task cache size increases, but higher associativity consistently gives lower effective access latency. Overall, the effective access latency for the smallest (64 entries) and the largest (1024 entries) task caches differs by as much as 3.0 cycles per access, a significant difference of 3 times the effective access latency. As might be expected, this significant difference in effective access latency means a significant difference in performance. The performance difference in instructions per cycle for the smallest and largest task cache size is as much as 1.50, for more than a 50% difference

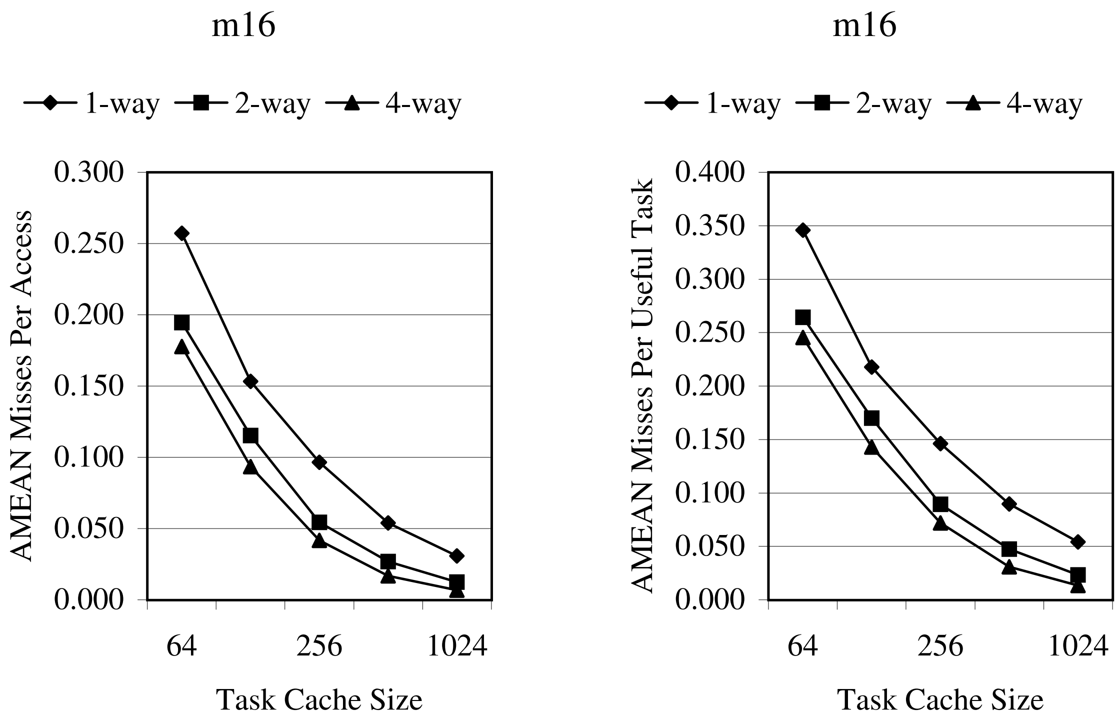


Figure 5.39: SPEC CINT95 misses per access and misses per useful task with 1-way, 2-way, and 4-way associative task caches for 16 processing units.

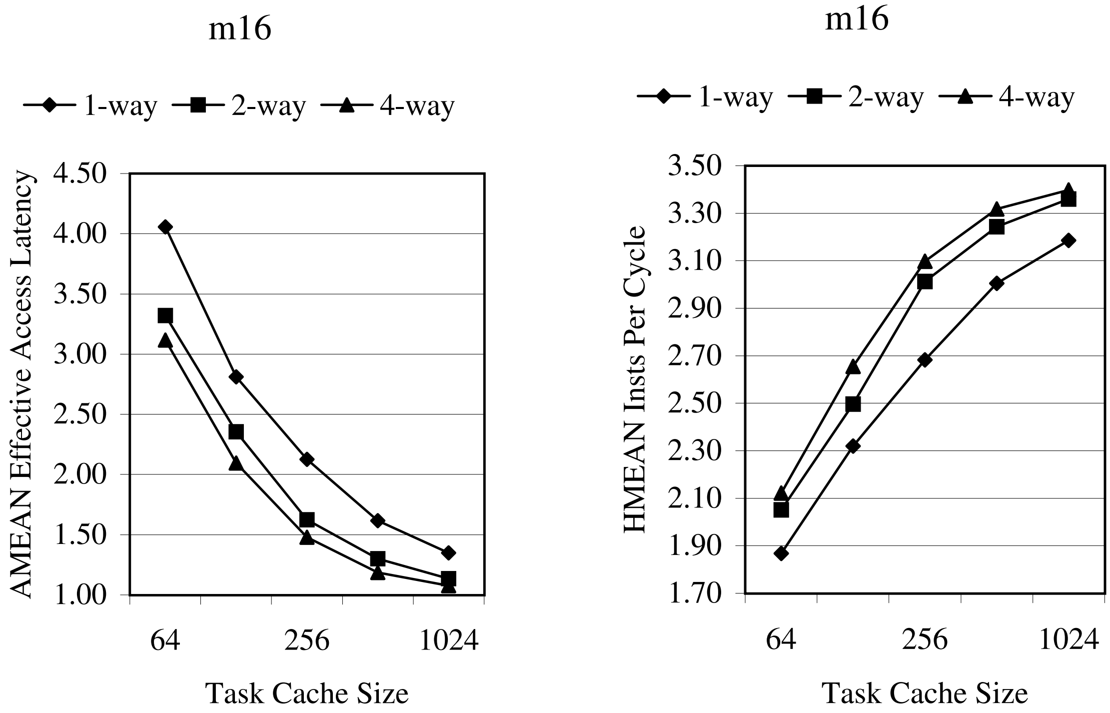


Figure 5.40: SPEC CINT95 effective access latency and performance with 1-way, 2-way, and 4-way associative task caches for 16 processing units.

in performance.

Figure 5.41 gives the speculative (on the left) and architectural (on the right) miss ratios with different task cache sizes and associativities for the SPEC CFP95 benchmarks using the 16 processing unit configuration. The relative behavior of the miss ratios for the task cache is similar to the behavior with respect to the SPEC CINT95 benchmarks. The main difference is that the magnitude of the relative differences is much smaller. While the SPEC CINT95 miss ratios differ as much as 0.05 misses per access in going from 1-way to 2-way associativity, the SPEC CFP95 miss ratios differ by not much more than 0.01 misses per access for the same change; the difference between 2-way and 4-way is almost insignificant. Moreover, the magnitude of the difference between the architectural and speculative ratios likewise is not nearly as significant for the floating point as for the integer benchmarks. Overall, the miss ratio for the smallest (64 entries) and the largest (1024 entries) task caches differ by only around 0.06 misses per access, a much less significant difference with respect to the overall miss ratio for the floating point than for the integer benchmarks.

Figure 5.42 gives the effective access latency (on the left) and the performance (on the right) with different task cache sizes and associativities for the SPEC CFP95 benchmarks using the 16 processing unit configuration. The floating point benchmark behavioral trends are similar to those observed for the integer benchmarks. However, the magnitude of the difference in effective access latency as well as in performance is much smaller for the SPEC CFP95 than for the SPEC CINT95. Among the different associativities, there is very slight difference in going from 1-way to 2-way associativity (not much more than 0.1 cycles per access) and an insignificant difference in going from 2-way to 4-way associativity. For the most part, the associativity becomes less of an influence as the task cache size increases; the effective access latency differences are so small that it is difficult to distinguish between them. Overall, the effective access latency of the smallest (64 entries) and the largest (1024 entries) task caches differs by less than 1.0 cycles per access, a much less significant difference for the floating point as compared to the integer benchmarks. Likewise, the performance difference in instructions per cycle between the smallest and largest cache sizes is not much greater than 0.20 instructions per cycle or less than a 3% difference in performance.

5.8.2.4 Real versus Ideal

In order to put the results of this task cache study in perspective, it is worthwhile to consider whether further improvements in task cache operation (aside from changing the hit latency) might provide further improvements in overall processor performance. An appropriate means of making such a determination is to compare the difference in overall processor performance using a real task cache (like the ones already studied) and using an ideal task cache. An ideal task cache has the same hit latency as a real task cache. However, unlike a real task cache which obviously has finite resources, it has infinite storage capacity (never has a miss), and it has infinite bandwidth (never has an address or a data conflict).

Figure 5.43 compares the performance on the SPEC CINT95 benchmarks of real 2-way associative task caches with 64, 128, 256, 512, or 1024 entries to that of an ideal task cache for processor configurations of 2, 4, 8, or 16 processing units running a single task per processing unit (one-at-a-time). Just beneath this figure, Figure 5.44 provides the same comparison on

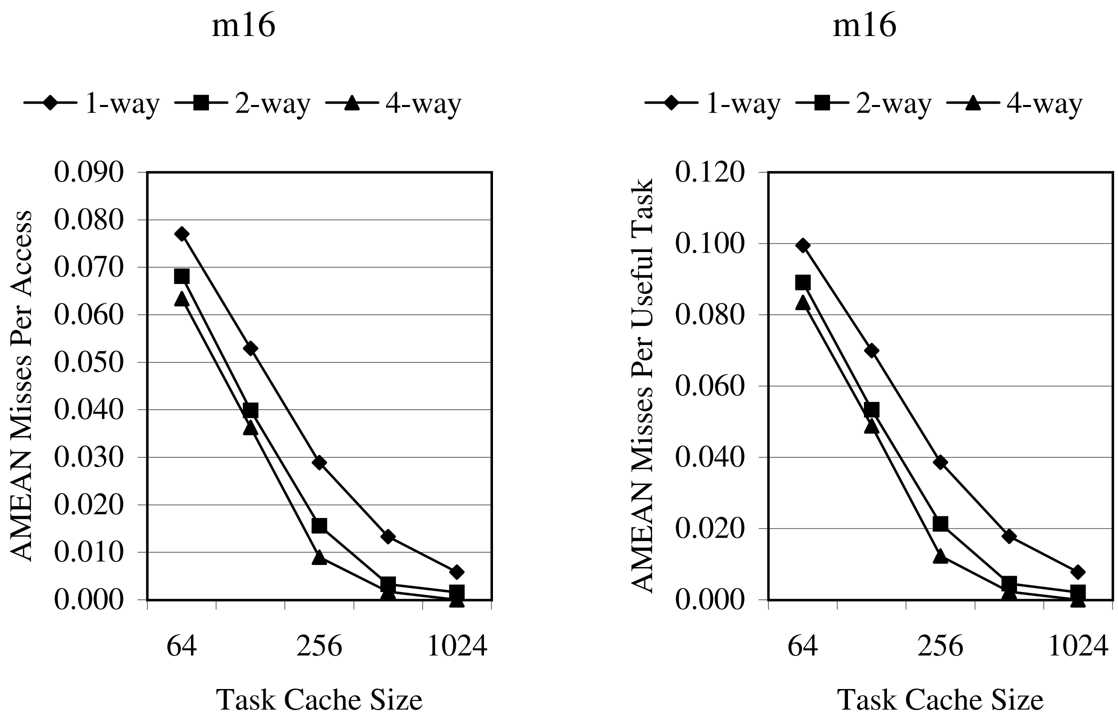


Figure 5.41: SPEC CFP95 misses per access and misses per useful task with 1-way, 2-way, and 4-way associative task caches for 16 processing units.

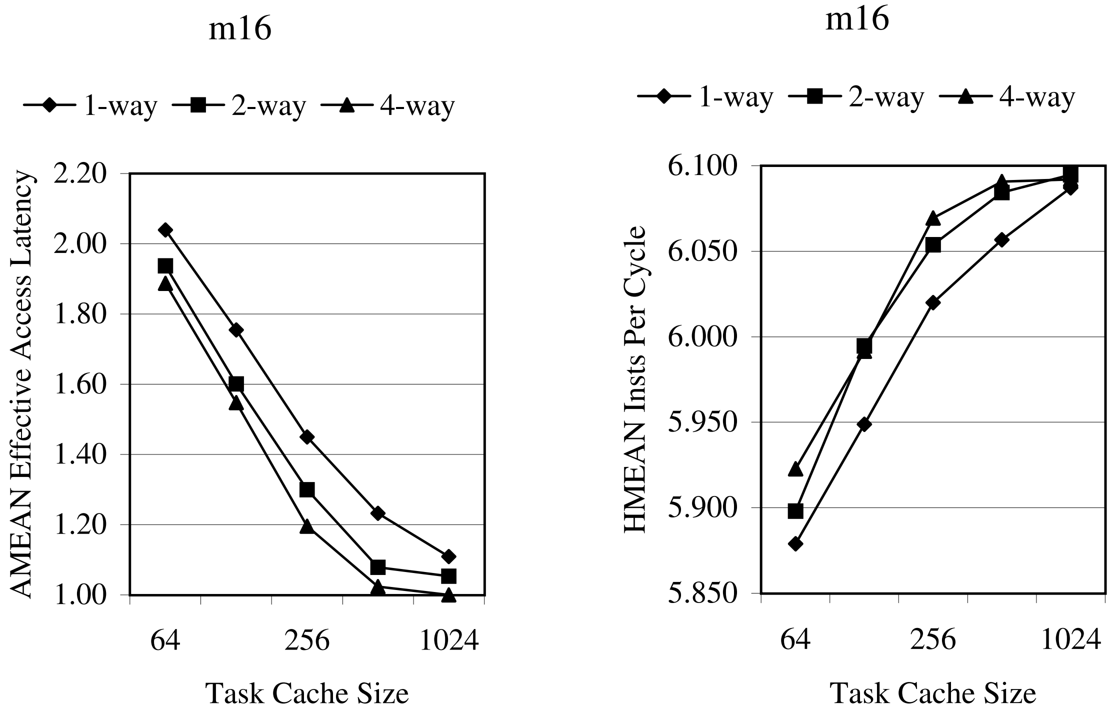


Figure 5.42: SPEC CFP95 effective access latency and performance with 1-way, 2-way, and 4-way associative task caches for 16 processing units.

the SPEC CFP95 benchmarks. The integer benchmarks show definite performance increases up to the biggest real task cache considered. However, these improvements diminish rapidly as the task cache size is doubled. As a result, an ideal task cache offers no significant performance advantage over the biggest real task cache for SPEC CINT95. The floating point benchmarks show no significant performance differences among all of the task caches considered. As a result, an ideal task cache offers no discernible performance advantage over even the smallest real task cache for SPEC CFP95.

5.8.3 Instruction Cache Study

Having considered the task cache aspect of the instruction memory evaluation, it is now necessary to turn to the instruction cache aspect. Because the intent of this part of the instruction memory evaluation is to focus only on the instruction cache, the other processor components, along with the task cache, are held constant in the standard configuration given in Table 3.11 of Chapter 3. The instruction cache configurations are varied as detailed next.

5.8.3.1 Configurations

The instruction cache is configured as both shared and private organizations. Each of these organizations is configured with different sizes. In addition, the shared organization is configured with different access times (on a hit) of 1, 2, 3, and 4 cycles (pipelined where the pipeline is flushed on a miss) to study this effect (due to the interconnect) on overall processor performance. The private organization is configured with a 1 cycle access time (on a hit) for all of the studies performed. The block size (32 bytes) and the replacement policy (least recently used) are fixed in this evaluation for both organizations, but the associativity is varied among 1-way, 2-way, and 4-way. For the shared organization, the instruction cache size is varied among 16K-byte, 32K-byte, 64K-byte, and 128K-byte overall for all numbers of processing units (with the storage divided equally in a like number of banks). For the private organization, the instruction cache size is varied among 1K-byte, 2K-byte, 4K-byte, 8K-byte, 16K-byte, 32K-byte, and 64K-byte for each individual processing unit depending on the overall number of processing units (with the total storage for all processing units not less than the smallest shared organization). Further, a hybrid organization combining private and shared aspects was studied. For all organizations, a range of processing unit configurations are used. In particular, the number of processing units is varied among 2, 4, 8, and 16, as well as among 4 processing units running 2 and 4 tasks one-at-a-time and all-at-once. Using these configurations, the instruction cache results are gathered in two steps.

First, the instruction cache evaluation focuses in general on the impact of different sizes among the processing unit configurations for both the shared and private organizations. For this evaluation, only 2-way associative instruction caches are used (the results for the other associativities are similar, though shifted higher or lower as seen in the second part of the evaluation). Only processing unit configurations that run a single task per processing unit are discussed. The overall differences among configurations are not that significant, and the multiple tasks per processing unit are similar to the results for the single task per processing unit configurations. This part of the instruction cache evaluation focuses on the behavior of

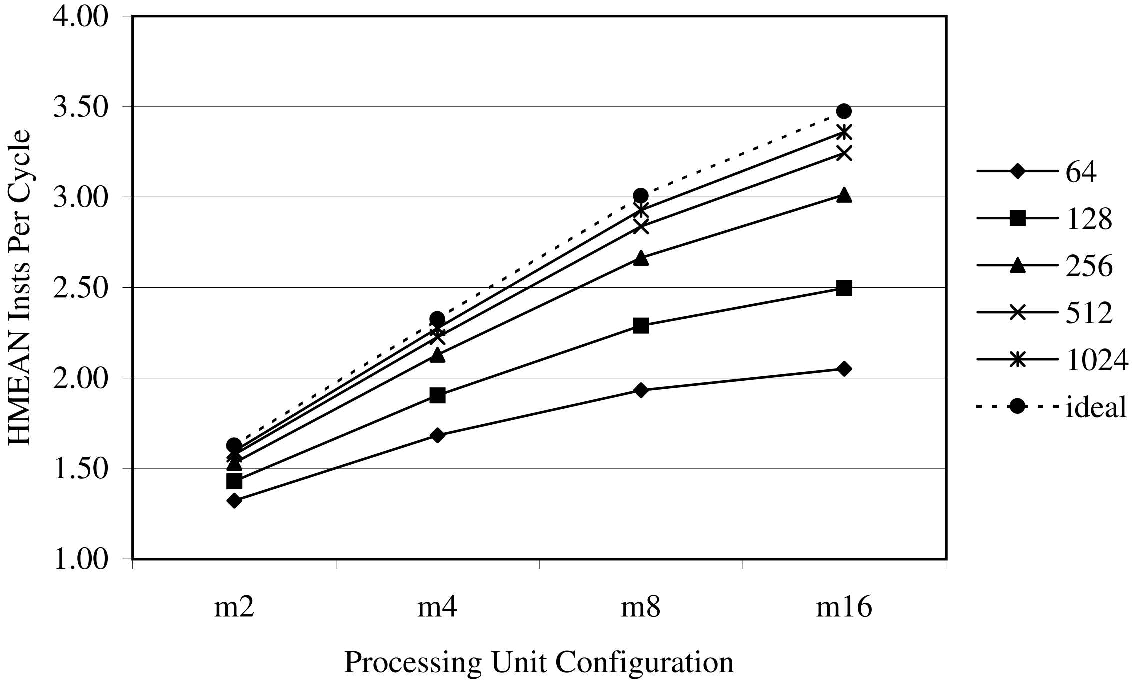


Figure 5.43: SPEC CINT95 performance of real 2-way associative compared to ideal task caches for 2, 4, 8, and 16 processing units.

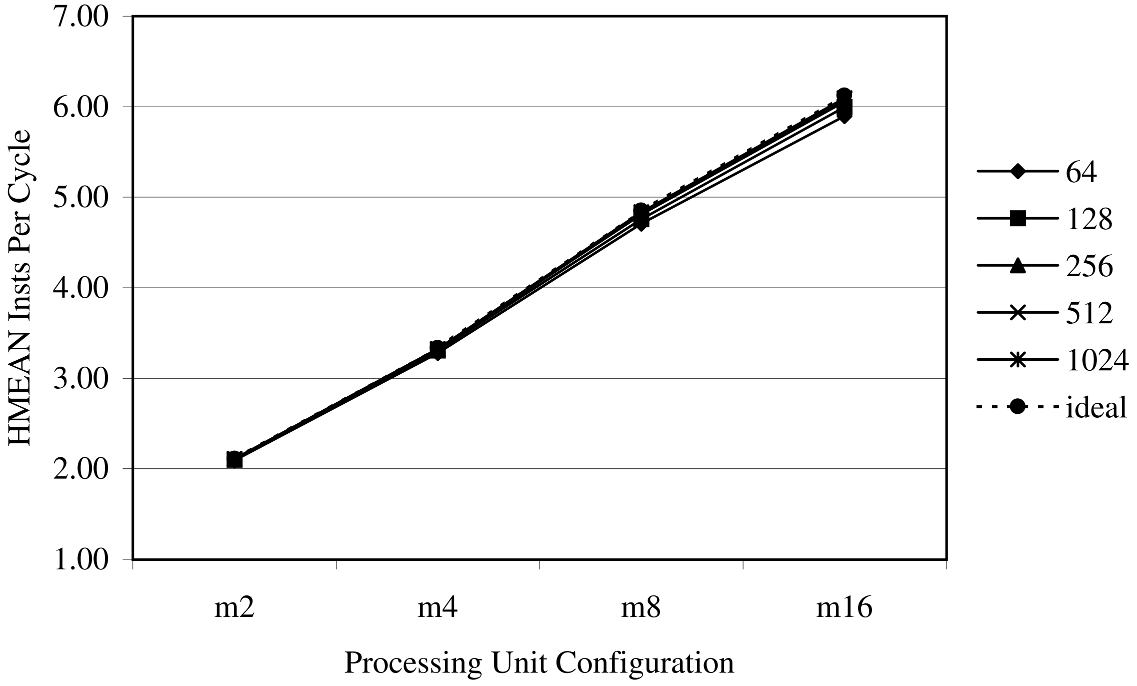


Figure 5.44: SPEC CFP95 performance of real 2-way associative compared to ideal task caches for 2, 4, 8, and 16 processing units.

one shared instruction cache configuration, 64K-byte with 2-way associativity, for different access latencies. In addition, the hybrid organization is considered as an alternative that addresses hit latency concerns with regard to the shared organization and miss latency concerns with regard to the private organization. Second, the instruction cache evaluation focuses in specific detail on the behavior with different sizes and associativities for only one processing unit configuration, 16 processing units, using both the shared and private organizations. The relative differences for all metrics with changing cache sizes and associativities is the same for each processing unit configuration.

5.8.3.2 General Results

Figure 5.45 gives the speculative (on the left) and architectural (on the right) miss ratios with different shared instruction cache sizes and 2-way associativity for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Beneath it, Figure 5.46 gives the same information for private instruction caches. The speculative miss ratio, misses per access, varies a reasonably significant amount from the smallest (16K-byte) to the largest (128K-byte) shared instruction cache organizations, but no more than 0.05 misses per access. In contrast, the speculative miss ratios for private cache organizations varies an extremely significant amount from the smallest (1K-byte) to the largest (64K-byte), at least 0.30 misses per access. Note that, a 16 processing unit configuration with a private cache organization uses a total of 16K-byte of storage for 1K-byte private instruction caches. Comparing the speculative miss ratio of this private organization to a shared organization of 16K-byte, the miss ratio differs significantly by around 0.25 misses per access. The difference between the organization does narrow considerably as bigger caches are used for the private organization. For instance, a 16 processing unit configuration that uses 64K-byte private caches has a lower speculative miss ratio than a shared 64K-byte cache, by 0.05 misses per access. However, in this case the private organization uses 16 times as much storage, a total of 1M-byte of storage, while only 2 times as much shared storage, 128K-byte, gives a comparable speculative miss ratio.

Among the different processing unit configurations, the speculative miss ratios for both the shared and private organizations do not change significantly, not by much more that 0.01 misses per access, as shown in Figure 5.45 for integer and Figure 5.47 for floating point. Nonetheless, the trends indicate that configurations with more processing units have lower miss ratios than those with less processing units for the shared organizations. This behavior is partly due to the effect of speculative execution as described for the task cache and partly due to the effect of combining that occurs for the multiple banks of the shared cache, which increases with the number of processing units and ranges from around 8% of accesses for 2 processing unit configurations to 14% of accesses for 16 processing unit configurations. It is difficult to separate the two effects, since some of the combining occurs for speculative accesses that are not useful. However, some of these speculative accesses likely ensure that blocks remain in the cache for later accesses that are useful. For private organizations, the trends do not indicate any difference among processing unit organizations for the speculative miss ratios, except for the 2 processing unit organization, which has a lower miss

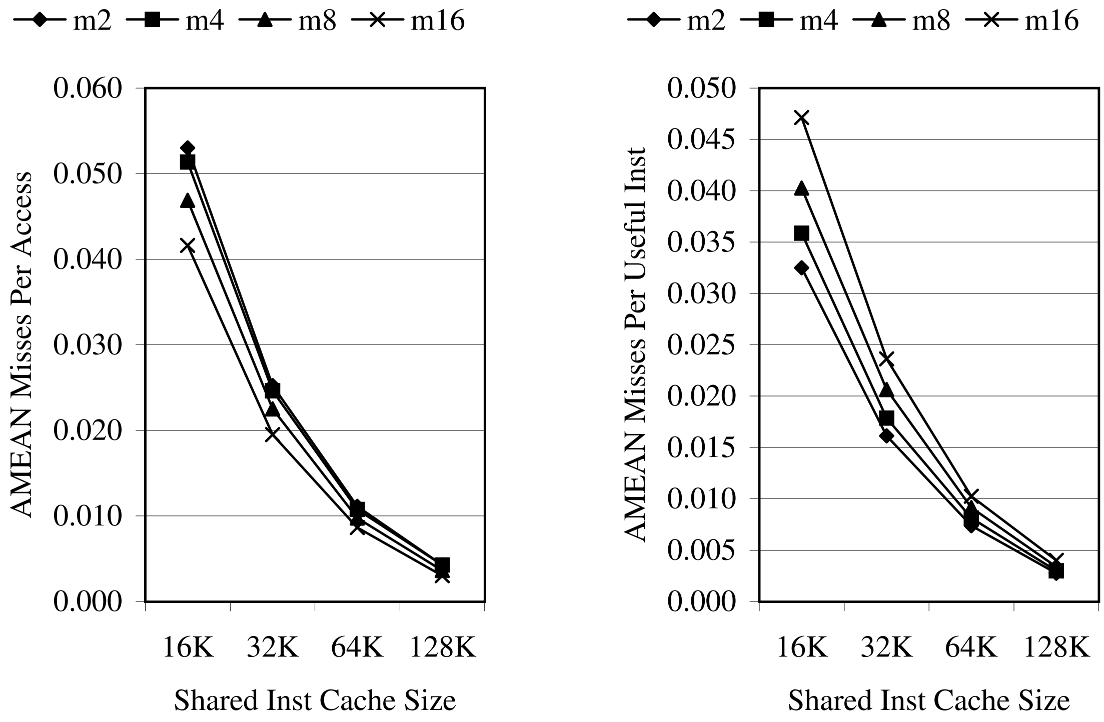


Figure 5.45: SPEC CINT95 misses per access and misses per useful instruction with 2-way associative shared instruction caches for 2, 4, 8, and 16 processing units.

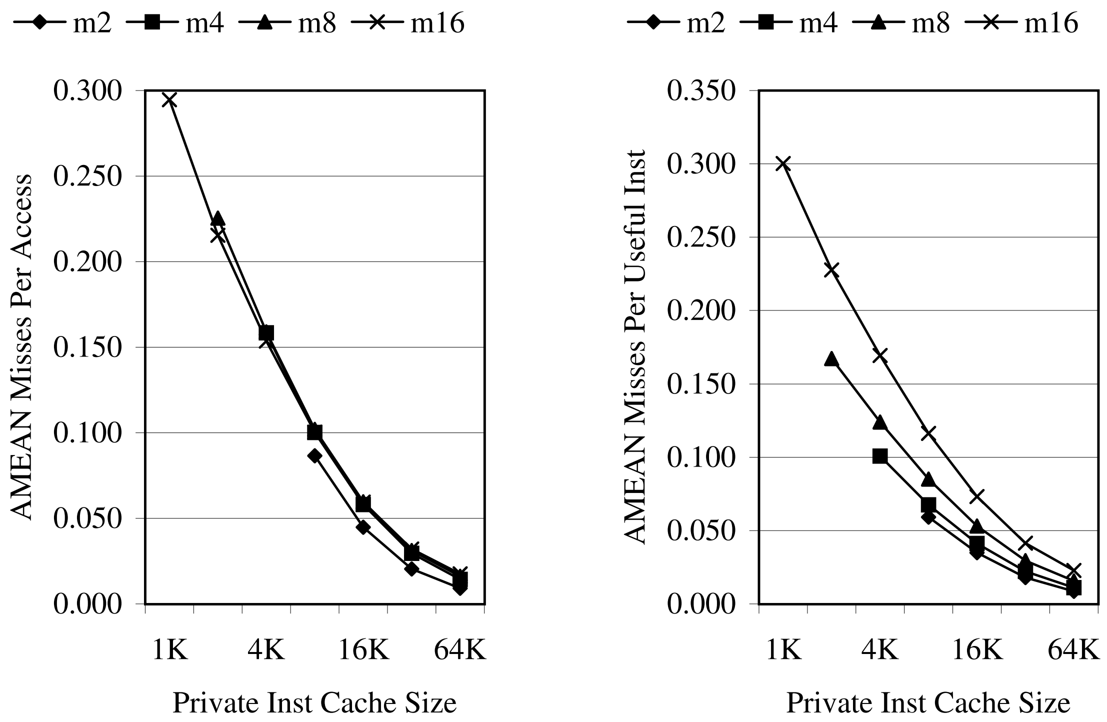


Figure 5.46: SPEC CINT95 misses per access and misses per useful instruction with 2-way associative private instruction caches for 2, 4, 8, and 16 processing units.

ratio. Though the use of read snarfing on instruction cache misses does allow misses by different private caches to combine, it does not actually change the observed miss ratio, since instructions are only provided with lower latency to caches that have already missed (but not accessed the next level of the memory hierarchy). For the 2 processing unit configuration, the use of only 2 private caches limits the access spreading enough that it nearly matches the speculative miss ratio of the shared caches.

The use of the speculative miss ratios in comparison to the architectural miss ratio is somewhat different for the instruction and task cache studies. For the instruction cache results, the difference in absolute terms between the speculative and architectural miss ratios is not apparent as for the task cache results. For the task cache results, each access provides a single task descriptor, so the absolute differences may be compared when dividing by the number of useful tasks. Yet, for the instruction cache results, each access provides multiple instructions (up to four), so dividing by the number of useful instructions makes it problematic to compare absolute differences. (Yet it is the only convenient architectural feature to use as a divisor, since the number of useful instruction accesses is hard to quantify.) Nevertheless, the trends remain the same, even if the absolute differences do not. That is, increases in the number of processing units decreases the speculative miss ratio, but increases the architectural miss ratio. This trend is the same as that observed for the task caches. Though the absolute differences are difficult to compare, the relative differences are not. In particular, the relative differences are more significant for the architectural miss ratios than for the speculative miss ratios. Furthermore, the relative differences for the private cache organizations far exceed those for the shared cache organizations, especially for the smaller cache sizes. However, the differences decline as the cache size doubles, and the overall miss ratio decreases for both shared and private organizations.

Figure 5.47 gives the speculative (on the left) and architectural (on the right) miss ratios with different shared instruction cache sizes and 2-way associativity for the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Beneath it, Figure 5.48 gives the same information for private instruction caches. The basic trends are the same for the floating point as for the integer benchmarks. However, there are two notable differences. First, the absolute difference in speculative miss ratios between the shared and private organizations is much lower for the floating point than for the integer benchmarks as the cache size halves. For the integer programs, the absolute difference with a 16 processing unit configuration for the smallest shared cache size (16K-byte) and the smallest private cache size (1K-byte and a total of 16K-byte) is more than 0.25 misses per access. In contrast, for the floating point programs, the absolute difference in the same case is only about 0.12 misses per access. Second, the differences among processing unit configurations is far more significant for the floating point benchmarks than for the integer benchmarks with both shared and private organizations. Nonetheless, the architectural miss ratios show the opposite behavior. In contrast to the integer benchmark, there is relatively little difference in the miss ratios for different processing unit organizations for the floating point benchmarks.

Figure 5.49 gives the effective access latency (on the left) and performance (on the right) with different shared instruction cache sizes and 2-way associativity for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Just beneath, Figure 5.50

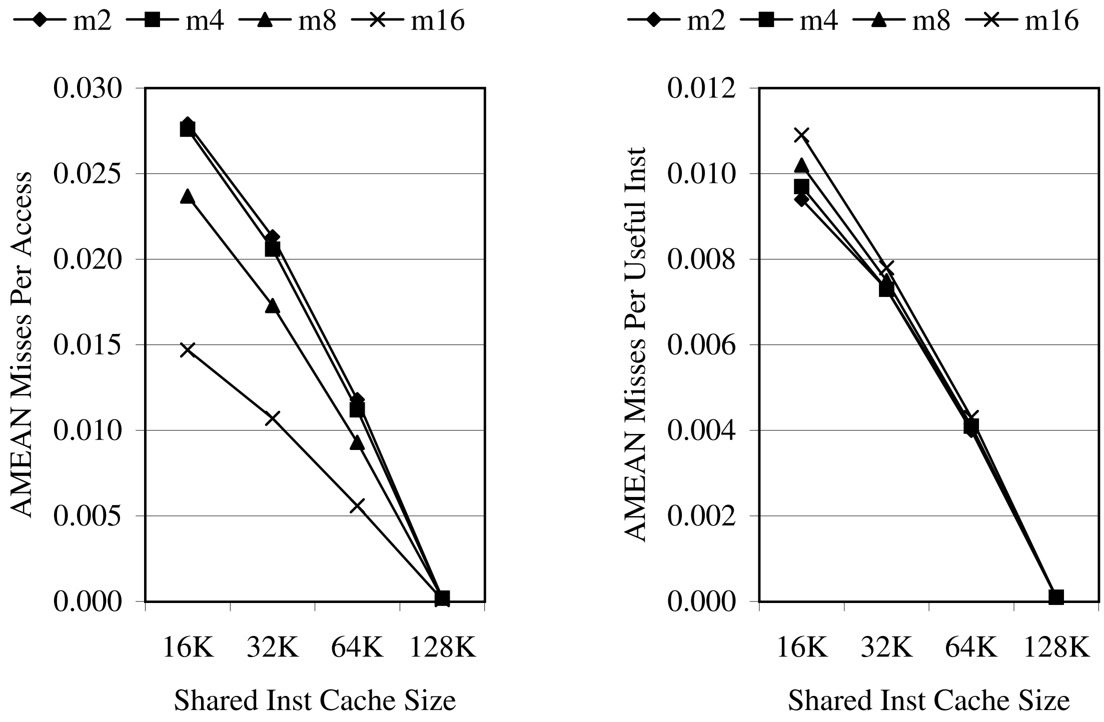


Figure 5.47: SPEC CFP95 misses per access and misses per useful instruction with 2-way associative shared instruction caches for 2, 4, 8, and 16 processing units.

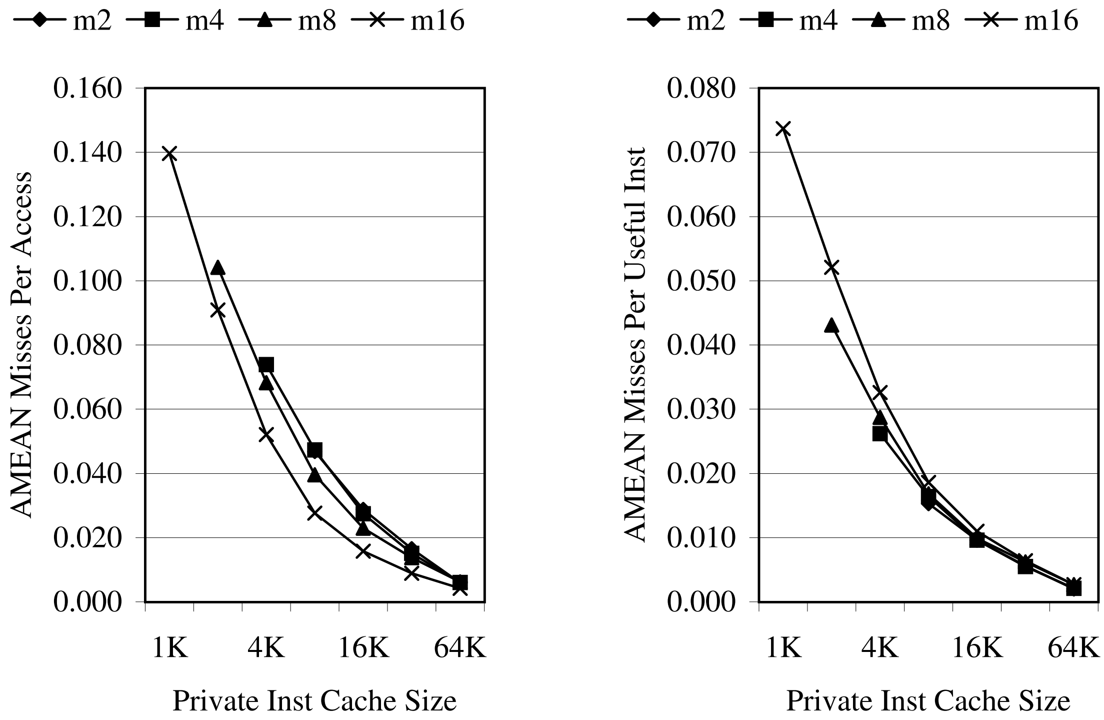


Figure 5.48: SPEC CFP95 misses per access and misses per useful instruction with 2-way associative private instruction caches for 2, 4, 8, and 16 processing units.

gives the same information for private instruction caches. In addition to these results for the integer benchmarks, the same results are given for the floating point benchmarks. Figure 5.51 gives the effective access latency (on the left) and performance (on the right) with different shared instruction cache sizes and 2-way associativity for the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Just beneath, Figure 5.52 gives the same information for private instruction caches.

The effective access latency with shared instruction cache organizations for the integer and the floating point benchmarks are comparable in terms of the absolute time in cycles as well as in terms of the trends among different processing unit organizations. Moreover, the trends between integer and floating point benchmarks for performance in instructions per cycle are very similar (though the absolute values are different) for the shared instruction cache organizations. In particular, the performance differences between the smallest (16K-byte) and largest (128K-byte) shared caches does not exceed 0.50 instructions per cycle. However, the performance differences increase as the number of the processing units increase.

The effective access latency with private instruction cache organizations for the integer and floating point benchmarks are similar in terms of the trends, but the absolute time in cycles is not always similar for integer and floating point. For shared cache sizes and individual private cache size of 16K-byte or more, the effective access latency is held in check and is comparable between the integer and floating point benchmarks. However, for shared cache sizes and individual private cache size of less than 16K-byte, the effective access latency degrades rapidly, but much more so for the integer than for the floating point benchmarks. In addition, the rapid degradation for integer benchmarks is more severe for configurations using more processing units.

It is worth noting that the performance in instructions per cycle falls off rapidly for the private instruction cache organizations much more severely than for the shared instruction cache organizations. In addition, this decrease is much more severe for integer than for floating point benchmarks. Furthermore, the integer benchmarks degrade quickly for all processing unit configurations, while the floating point benchmarks degrade more quickly as more processing units are used with the private organizations. Yet, this kind of severe degradation for different numbers of processing units does not seem to occur for shared caches, though some degradation does occur for both integer benchmarks (more so) and floating point benchmarks (less so). This behavior is obviously due to the much larger effective access latency of the private caches compared to shared caches.

Figure 5.53 gives the performance of a fixed 64K-byte and 2-way associativity shared instruction cache for hit latencies of 1, 2, 3, and 4 cycles (piplined where the pipeline is flushed on a miss) on the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. In addition to these results for the integer benchmarks, the same results are given for the floating point benchmarks. Figure 5.54 gives the performance of a fixed 64K-byte and 2-way associativity shared instruction cache for hit latencies of 1, 2, 3, and 4 cycles (pipelined where the pipeline is flushed on a miss) on the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Though the absolute numbers for the two groups of benchmarks are obviously quantitatively different, it is interesting to compare them qualitatively.

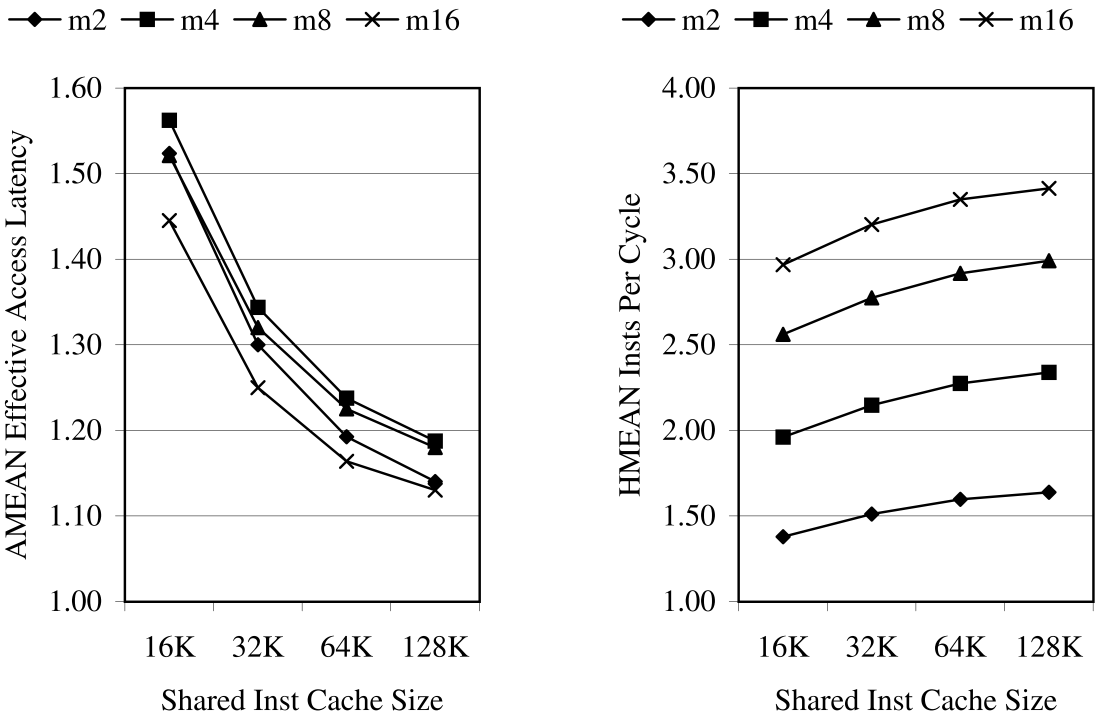


Figure 5.49: SPEC CINT95 effective access latency and performance with 2-way associative shared instruction caches for 2, 4, 8, and 16 processing units.

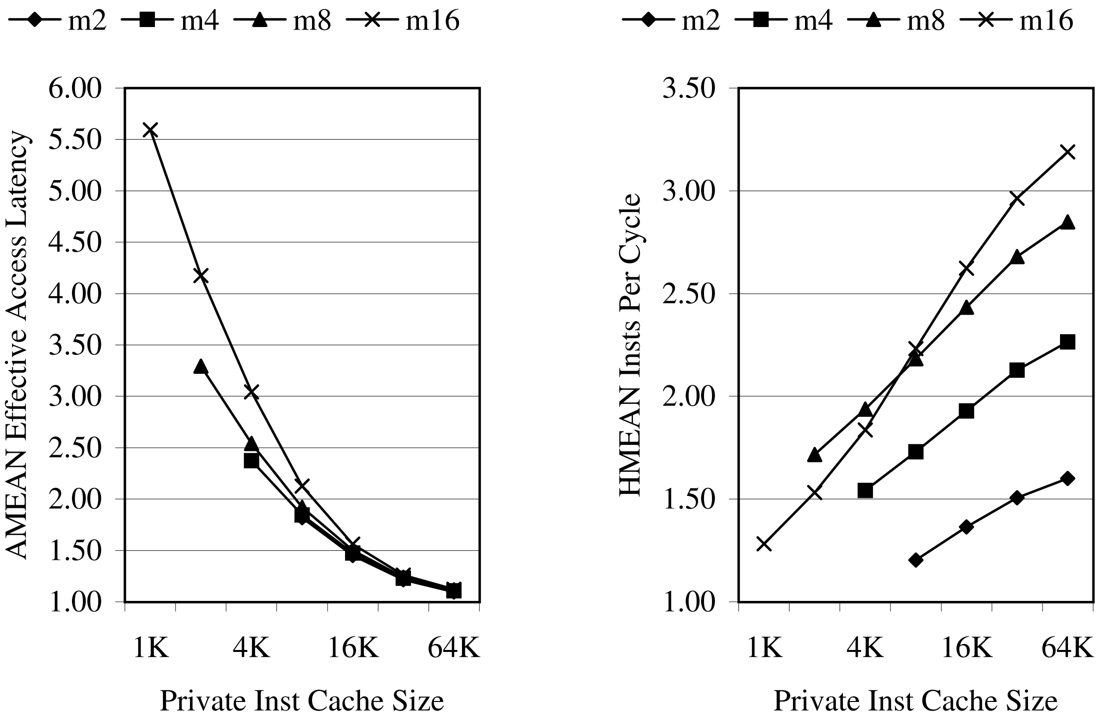


Figure 5.50: SPEC CINT95 effective access latency and performance with 2-way associative private instruction caches for 2, 4, 8, and 16 processing units.

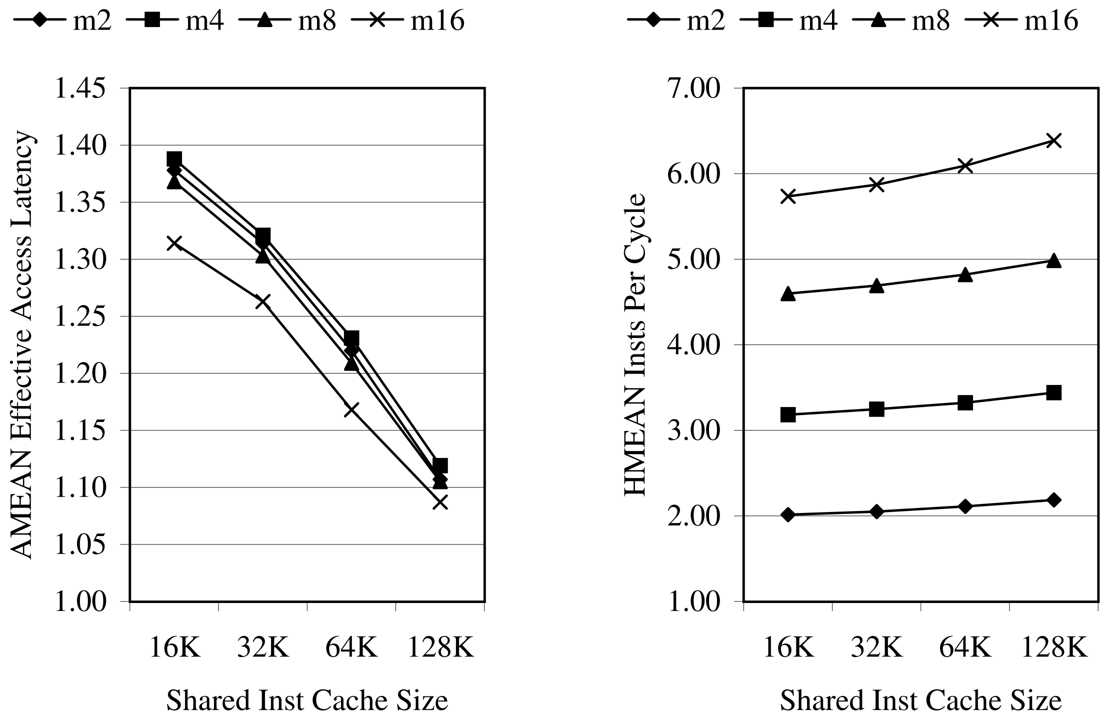


Figure 5.51: SPEC CFP95 effective access latency and performance with 2-way associative shared instruction caches for 2, 4, 8, and 16 processing units.

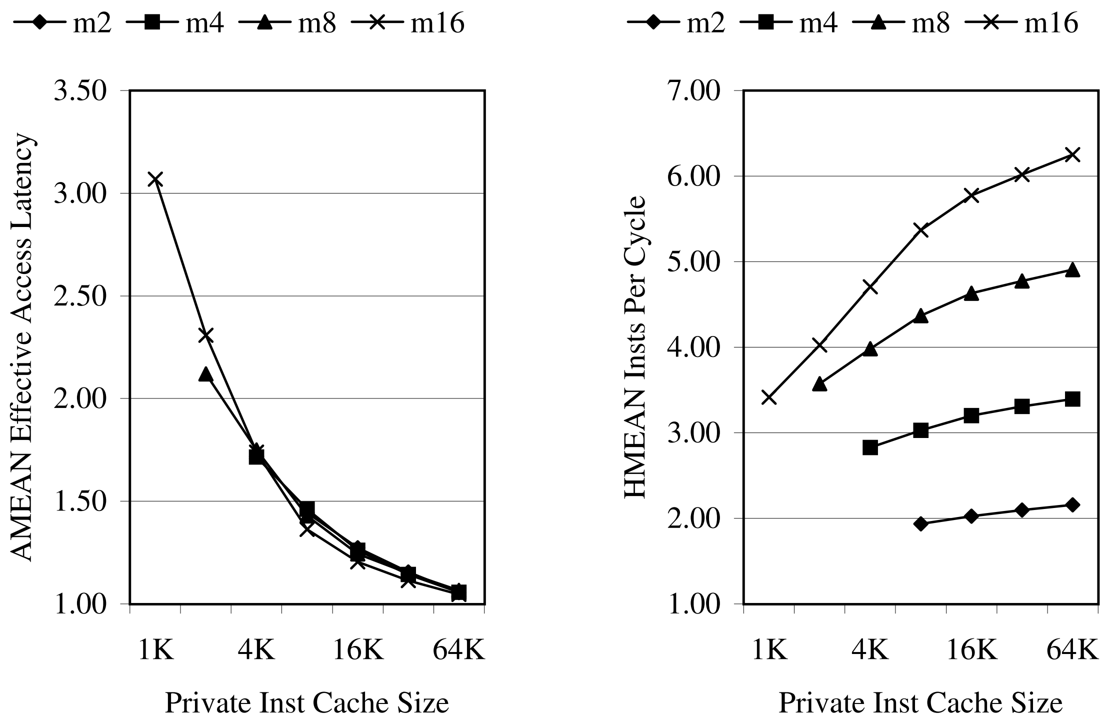


Figure 5.52: SPEC CFP95 effective access latency and performance with 2-way associative private instruction caches for 2, 4, 8, and 16 processing units.

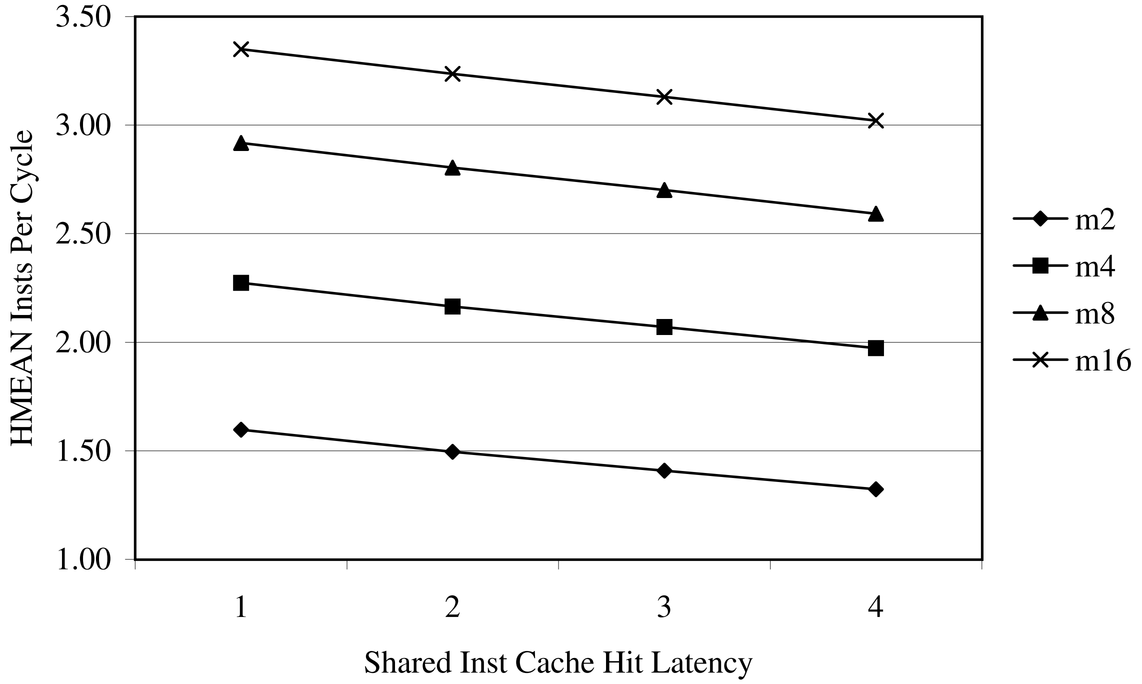


Figure 5.53: SPEC CINT95 performance with 64K-byte 2-way associative shared instruction cache for hit latencies of 1, 2, 3, and 4 cycles (pipelined where the pipeline is flushed on a miss).

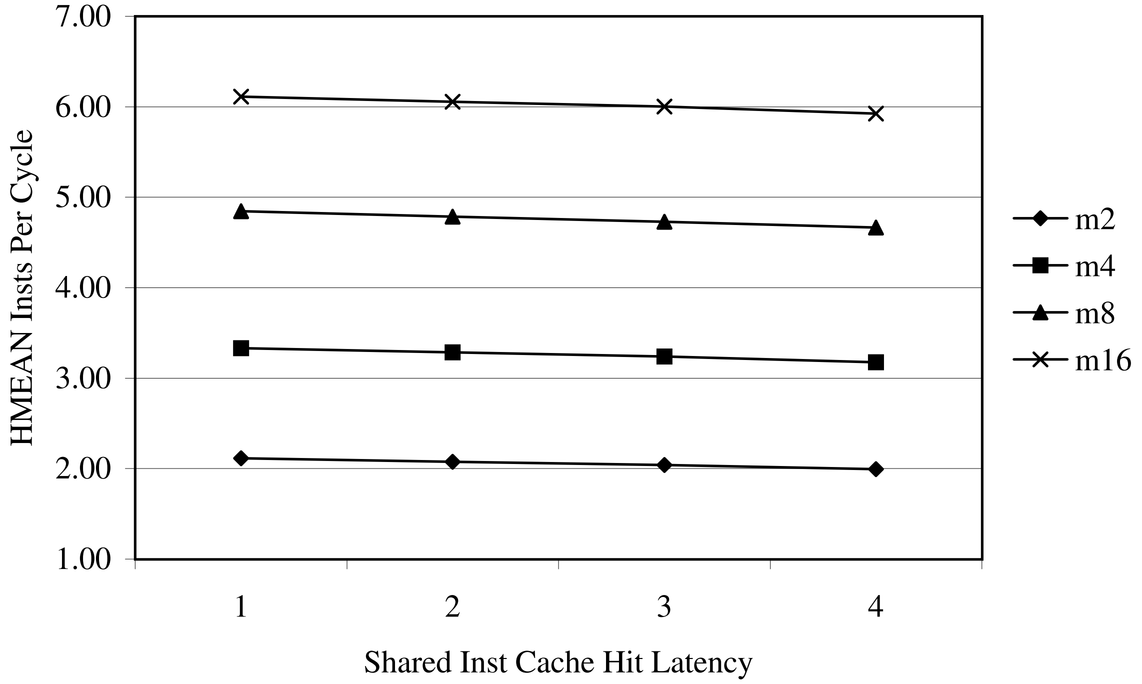


Figure 5.54: SPEC CFP95 performance with 64K-byte 2-way associative shared instruction cache for hit latencies of 1, 2, 3, and 4 cycles (pipelined where the pipeline is flushed on a miss).

As shown in Figures 5.53 and 5.54, increasing the hit latency of the shared caches does not have such a drastic impact on performance, even for latencies up to 4 cycles. However, there is a significant impact on performance for private caches with high effective access latencies (as seen earlier). The likely reason for this difference in behavior is that the access latency is for misses in the private cache case which stall individual processing units (providing no latency tolerance), while the access latency is for hits in the shared cache case which may be pipelined on the processing units (providing latency tolerance). For the private organizations many processing units may miss in their private caches for the same instructions, but read snarfing on misses does not help (even less for integer than for floating point judging by the effective access latencies) to remedy this situation.

As described above for Figures 5.53 and 5.54, the private instruction caches suffer a significant performance degradation compared to the shared instruction caches because of their high miss rates coupled with the high cost of servicing those misses. Furthermore, in order to reduce the high miss rates to overcome this performance degradation, it is necessary to use inordinate amounts of storage in the private instruction caches relative to shared instruction caches. Thus, the key to improving the performance of the private instruction caches is reducing their miss rate or reducing the cost of servicing their misses. Since reducing the private instruction cache miss rate using more storage is an untenable the option of due to the associated storage cost, it appears more fitting to instead consider reducing the cost of servicing their misses.

A natural means of accomplishing this goal is to consider a hybrid instruction cache design that interposes the small private instruction caches between the processing units and the shared instruction cache, rather than between the processing units and the next level of the cache hierarchy. This hybrid instruction cache does indeed use some additional storage, but this storage need not scale (on a per private cache basis) with the number of processing units and may in fact allow for a reduction in the size of the private instruction caches. In addition, this hybrid may allow the total bandwidth that must be provided by the shared instruction cache to be reduced in order to simplify its design. Such a design is worth considering because, even though shared instruction caches do provide good performance, it is unlikely that their designs will be able to provide the 1 cycle access with high bandwidth that gives the best performance. In particular, as already described, there is a small, but uniform drop in processor performance as the latency of the shared instruction cache hit latency is increased.

Figure 5.55 compares the performance on the SPEC CINT95 benchmarks of a 64K-byte 2-way associative shared instruction cache with 1, 2, 3, or 4 cycles hit latency (pipelined where the pipeline is flushed on a miss) to that of a hybrid of 4K-byte 2-way associative private instruction caches with 1 cycle hit latency plus a 64K-byte 2-way associative shared instruction cache with 4 cycles hit latency (pipelined where the pipeline is flushed on a miss). Likewise, Figure 5.56 provides the same comparison on the SPEC CFP95 benchmarks. The processor configurations are 2, 4, 8, or 16 processing units running a single task per processing unit (one-at-a-time). In terms of communication between the two parts of the hybrid instruction cache, the private instruction caches are connected to the shared instruction cache via a split-transaction bus. Moreover, the shared instruction cache is only capable of servicing one access per cycle. The total storage used for the hybrid instruction cache is 40K-byte for 2 processing units, 48K-byte for 4 processing units, 64K-byte for 8 processing units, or 128K-byte for 16 processing units.

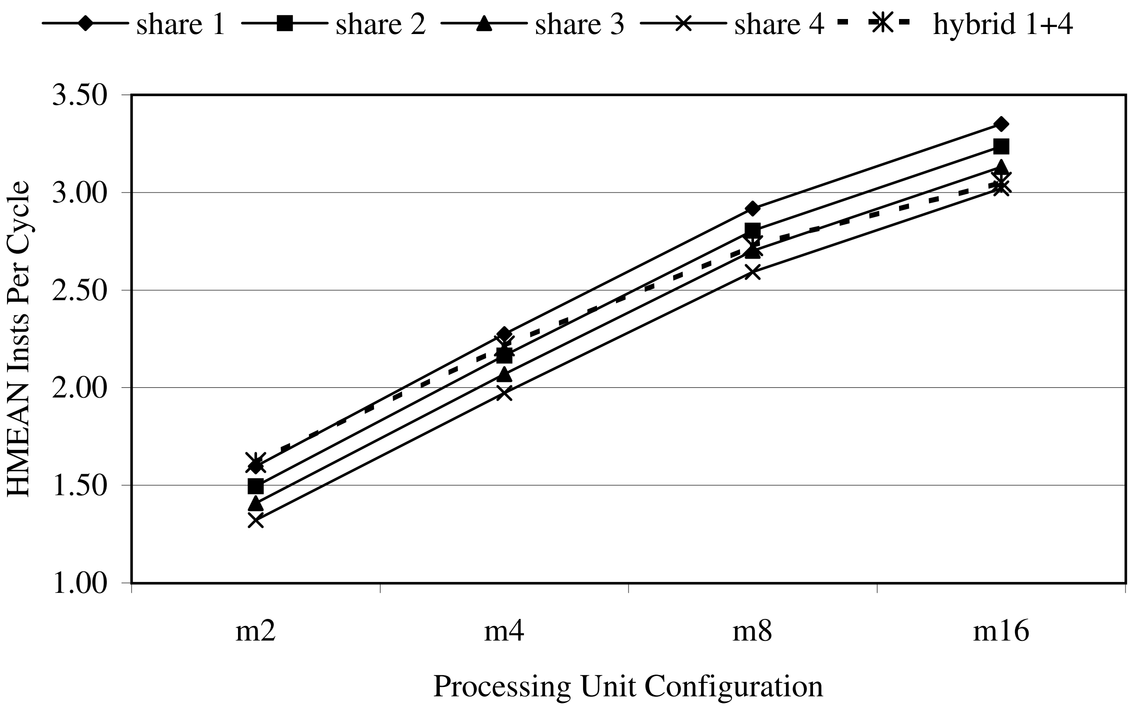


Figure 5.55: SPEC CINT95 performance of 64K-byte 2-way associative shared instruction cache with 1, 2, 3, or 4 cycles hit latency compared to hybrid of 4K-byte 2-way associative private instruction caches with 1 cycle hit latency plus 64K-byte 2-way associative shared instruction cache with 4 cycles hit latency (using pipelined caches where the pipeline is flushed on a miss).

For the integer benchmarks, the hybrid instruction cache provides the same performance as the 1 cycle hit latency shared instruction cache for the smallest, 2 processing unit configuration. However, for the biggest, 16 processing unit configuration, the hybrid instruction cache degrades to provide the same performance as the 4 cycle hit latency shared instruction cache. The configurations in between trace this degradation with the 4 processing unit configuration providing the same performance as the 2 cycle hit latency shared instruction cache and the 8 processing unit configuration provide the same performance as the 3 cycle hit latency shared instruction cache. Because the private instruction caches have relatively high miss rate and the shared instruction cache has relatively low bandwidth, an increase in demand from more processing units cannot be satisfied, and a decrease in performance results. The floating point benchmarks have similar behavior to the integer benchmarks, though it is more difficult to distinguish, because there are insignificant decreases in performance with increases in the shared instruction cache latency.

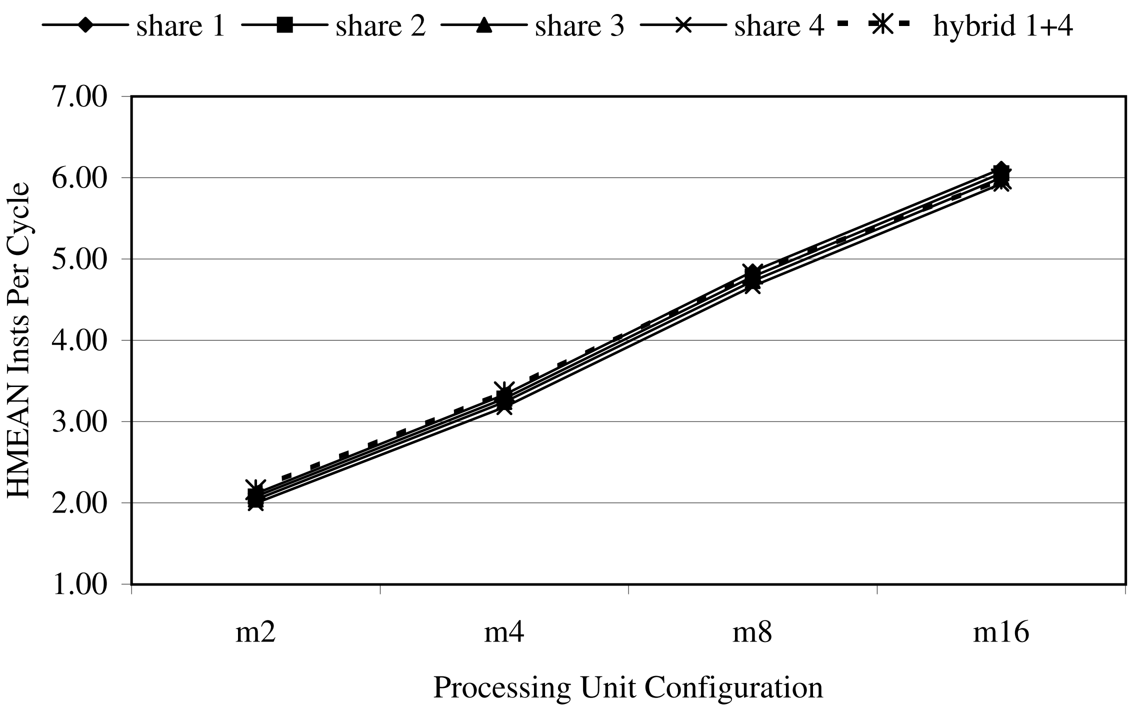


Figure 5.56: SPEC CFP95 performance of 64K-byte 2-way associative shared instruction cache with 1, 2, 3, or 4 cycles hit latency compared to hybrid of 4K-byte 2-way associative private instruction caches with 1 cycle hit latency plus 64K-byte 2-way associative shared instruction cache with 4 cycles hit latency (using pipelined caches where the pipeline is flushed on a miss).

5.8.3.3 Specific Results

Figure 5.57 gives the speculative (on the left) and architectural (on the right) miss ratios with different shared instruction cache sizes and associativities for the SPEC CINT95 benchmarks using the 16 processing unit configuration. Beneath it, Figure 5.58 gives the same information for private instruction caches. The speculative miss ratio, misses per access, and the architectural miss ratio, misses per useful instruction decrease steadily as the cache size is doubled for all associativities for both the shared and private organizations. Among the different associativities for the shared organizations, there is only a small difference going from 1-way to 2-way associativity (not much more than 0.01 misses per access) and an even smaller difference going from 2-way to 4-way associativity (not as much as half 0.01 misses per access). The private organizations show this same trend but the differences are somewhat bigger (by a factor of about 2 times), though it is difficult to tell because of the scale in the graph. Overall, the private organizations have much higher miss ratios for comparable amounts of total storage (as already discussed). The architectural miss ratios track the speculative miss ratios as only the divisor in the calculation of the ratio has changed, not the level of speculative execution since the number of processing units is held constant.

Figure 5.59 gives the effective access latency (on the left) and the performance (on the right) with different shared instruction cache sizes and associativities for the SPEC CINT95 benchmarks using the 16 processing unit configuration. Beneath it, Figure 5.60 gives the same information for private instruction caches. In accordance with the behavior of the miss ratios, the effective access latency decreases steadily as the instruction cache size is doubled for all associativities and for both shared and private organizations. Among the different associativities, there is a more significant difference going from 1-way to 2-way associativity for the private organization (nearly 0.5 cycles per access) than for the shared organizations (not more than 0.2 cycles per access). The difference going from 2-way to 4-way associativity is more or less insignificant for both shared and private organizations. Furthermore, the associativity becomes slightly less of an influence as the instruction cache sizes increase for both organizations, but higher associativity consistently gives lower effective access latency.

Overall, the effective access latency for the smallest (16K-byte for shared and 1K-byte for private) and the largest (128K-byte for shared and 64K-byte for private) instruction caches differs by not much more than 1.0 cycles per access for shared organizations and by as much as 5.0 cycles per access for private organizations. As might be expected, this significant difference in effective access latency between the shared and private organizations means a significant difference in performance. The performance difference in instructions per cycle for the smallest (16K-byte) and largest (128K-byte) shared instruction caches is not much more than 0.50 instructions per cycle. Yet, the difference for the smallest (1K-byte or 16K-byte) and largest (64K-byte) private instruction caches is nearly 2.00 instructions per cycle if the *total* storage of the smallest private caches (1K-byte each) is the same as the total storage of the smallest shared cache, and about 1.00 instructions per cycle if the *individual* storage of the smallest private caches (16K-byte each) is the same as the total storage of the smallest shared cache (16K-byte).

Figure 5.61 gives the speculative (on the left) and architectural (on the right) miss ratios

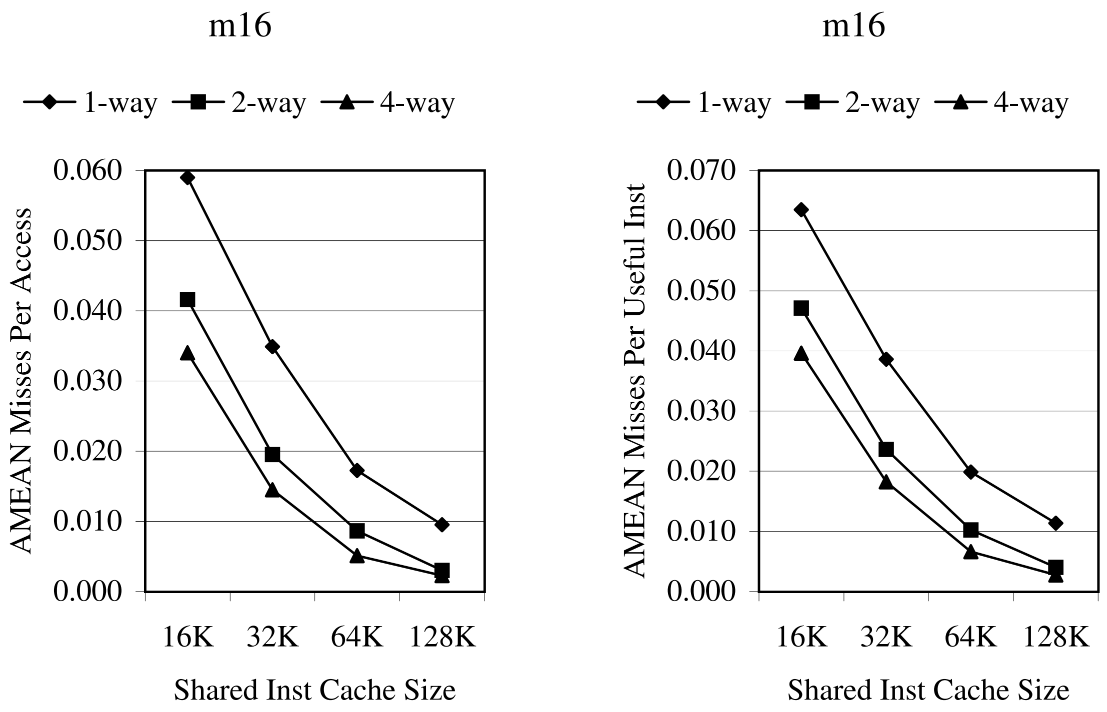


Figure 5.57: SPEC CINT95 misses per access and misses per useful instruction with 1-way, 2-way, and 4-way associative shared instruction caches for 16 processing units.

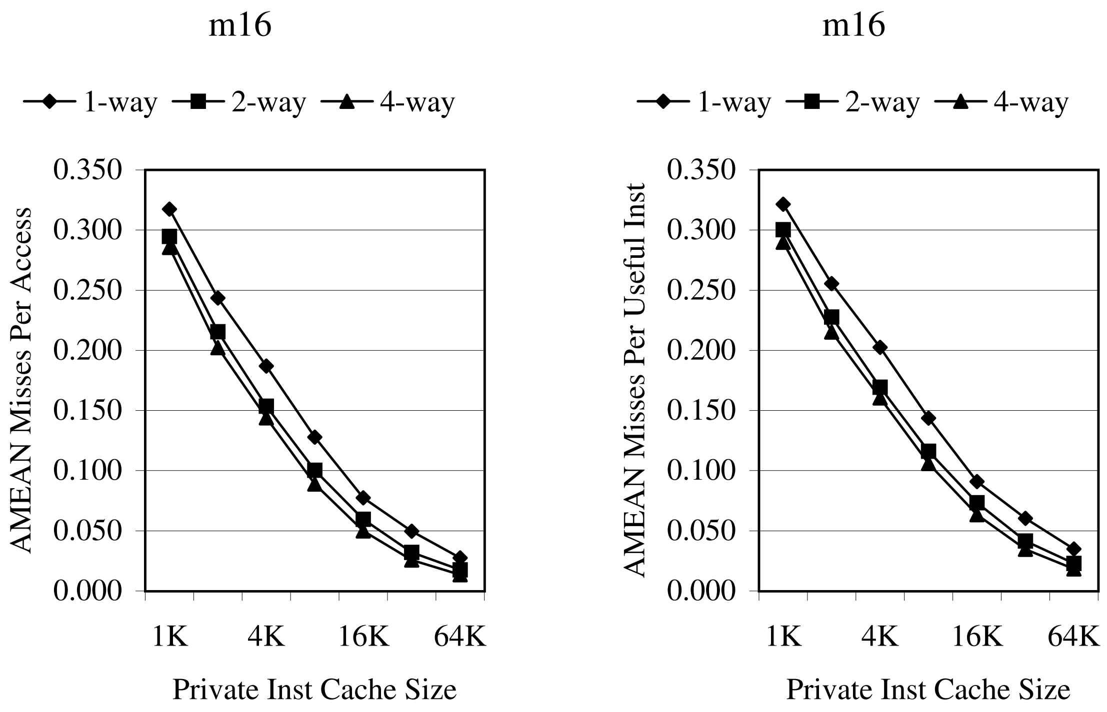


Figure 5.58: SPEC CINT95 misses per access and misses per useful instruction with 1-way, 2-way, and 4-way associative private instruction caches for 16 processing units.

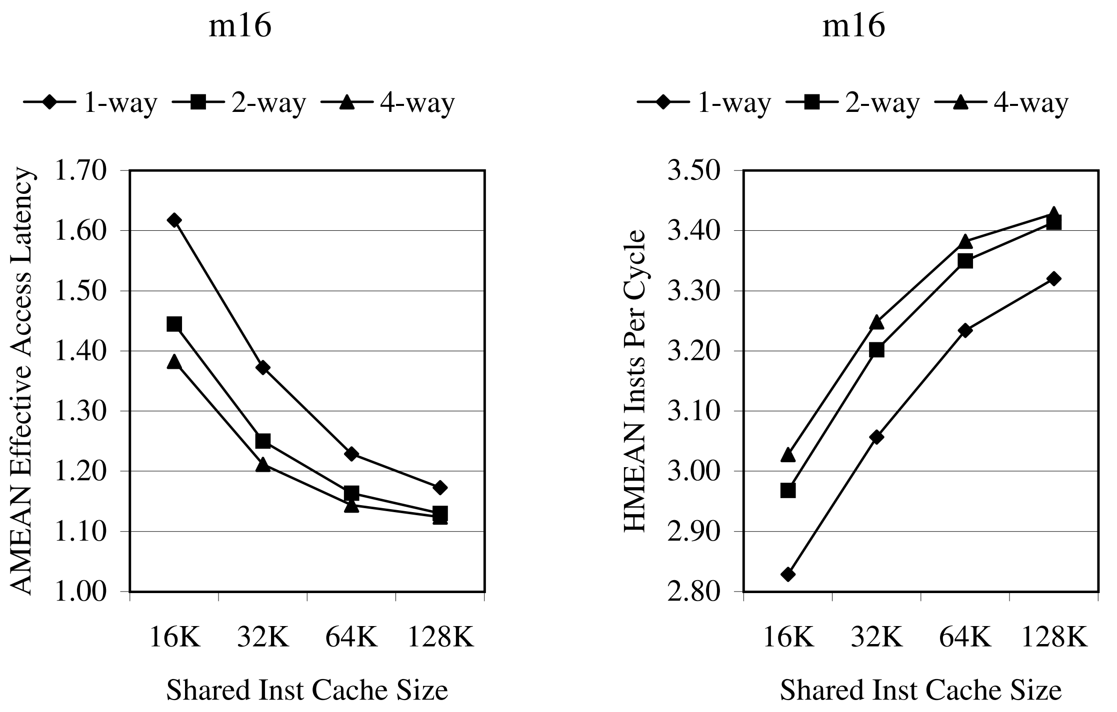


Figure 5.59: SPEC CINT95 effective access latency and performance with 1-way, 2-way, and 4-way associative shared instruction caches for 16 processing units.

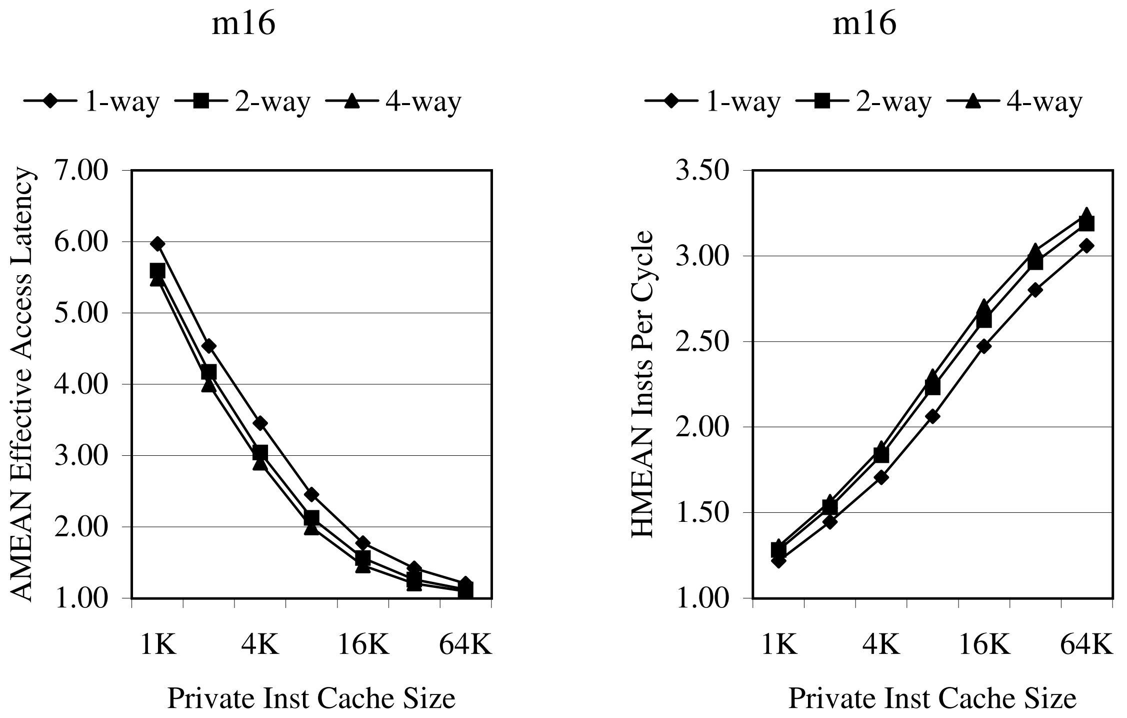


Figure 5.60: SPEC CINT95 effective access latency and performance with 1-way, 2-way, and 4-way associative private instruction caches for 16 processing units.

with different shared instruction cache sizes and associativities for the SPEC CFP95 benchmarks using the 16 processing unit configuration. Beneath it, Figure 5.62 gives the same information for private instruction caches. The relative behavior of the miss ratios for the shared instruction caches is somewhat different from the behavior with respect to the SPEC CINT95 benchmarks, but similar for the private instruction caches. For the shared organizations, the associativity has little or no effect except for the smallest (16K-byte) 1-way design; the 2-way and 4-way designs show almost no difference, though the 4-way design is always better. Noting the scale on the graph, the differences between the associativities for a given shared cache size is only around 0.001 misses per access. For the private organizations, the behavior with respect to the associativity for the floating point benchmarks is nearly the same as for the integer benchmarks. The main difference is that the magnitude of the relative differences is much smaller. Overall, the private organizations have much higher miss ratios once the individual cache sizes go below the overall shared cache sizes. Again, the architectural miss ratios track the speculative miss ratios as only the divisor in the calculation of the ratio has changed, not the level of speculative execution since the number of processing units is held constant.

Figure 5.63 gives the effective access latency (on the left) and the performance (on the right) with different shared instruction cache sizes and associativities for the SPEC CFP95 benchmarks using the 16 processing unit configuration. Beneath it, Figure 5.64 gives the same information for private instruction caches. In accordance with the behavior of the miss ratios, the effective access latency decreases steadily as the instruction cache size is doubled for all associativities and for both shared and private organizations. Among the different associativities, there is only a significant difference between the associativities for the smallest (16K-byte) 1-way shared cache. Otherwise, the difference between the associativities for the shared caches is negligible. Likewise, the difference between the associativities for the private caches is negligible except for the smallest (1K-byte) 1-way private cache. As for the integer results, the associativity becomes less of an influence as the instruction cache sizes increase for both organizations, but higher associativity consistently gives lower effective access latency (though the difference is difficult to discern).

Overall, the effective access latency for the smallest (16K-byte for shared and 1K-byte for private) and the largest (128K-byte for shared and 64K-byte for private) instruction caches differs by not much more than 0.3 cycles per access for shared organizations and by as much as 2.0 cycles per access for private organizations. Though this difference is much smaller than for the integer benchmarks in absolute terms (1.0 cycles per access for shared versus 5.0 cycles per access for private), it is still enough to produce a significant difference between the shared and private organizations in terms of performance. The performance difference in instructions per cycle for the smallest and largest shared instruction caches is no more than 1.00 instructions per cycle. However, the difference for the smallest (1K-byte or 16K-byte) and largest (64K-byte) private instruction caches is more than 3.00 instructions per cycle if the *total* storage of the smallest private caches (1K-byte each) is the same as the total storage of the smallest shared cache, and only about 1.00 instructions per cycle if the *individual* storage of the smallest private caches (16K-byte each) is the same as the total storage of the smallest shared cache (16K-byte).

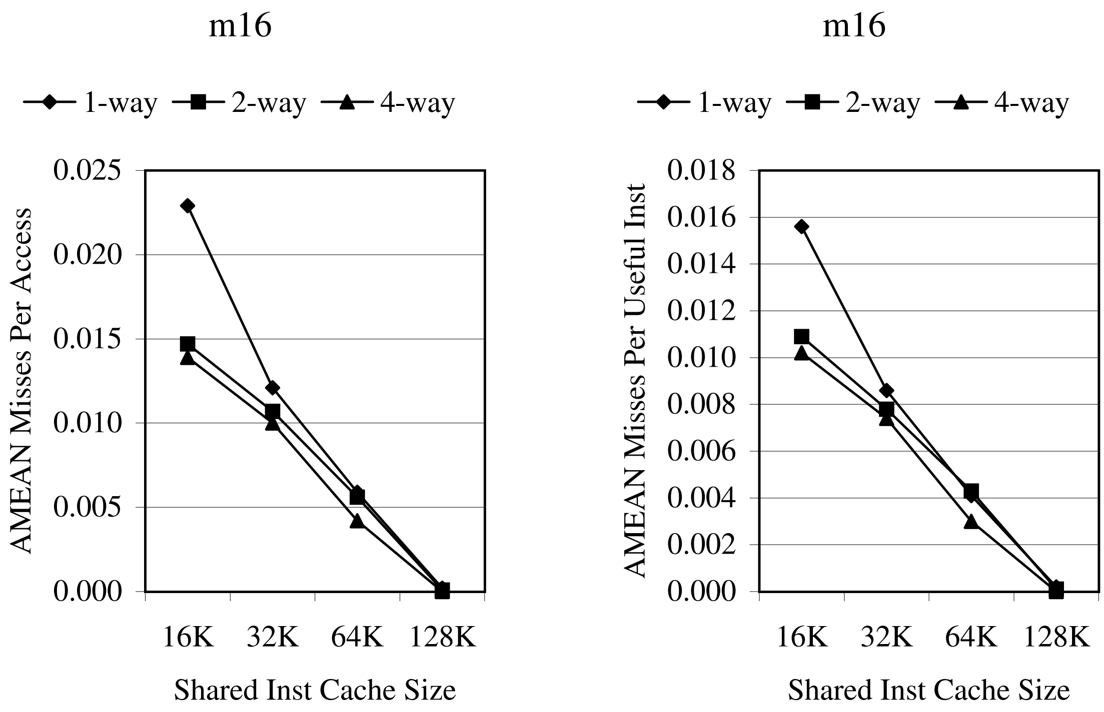


Figure 5.61: SPEC CFP95 misses per access and misses per useful instruction with 1-way, 2-way, and 4-way associative shared instruction caches for 16 processing units.

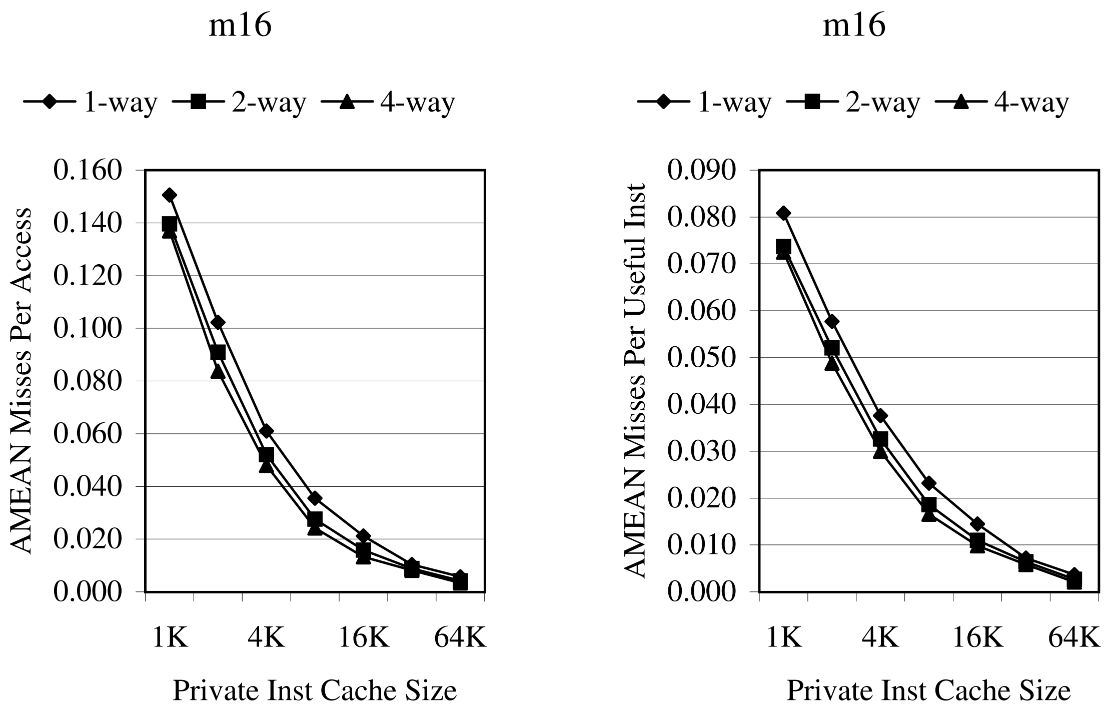


Figure 5.62: SPEC CFP95 misses per access and misses per useful instruction with 1-way, 2-way, and 4-way associative private instruction caches for 16 processing units.

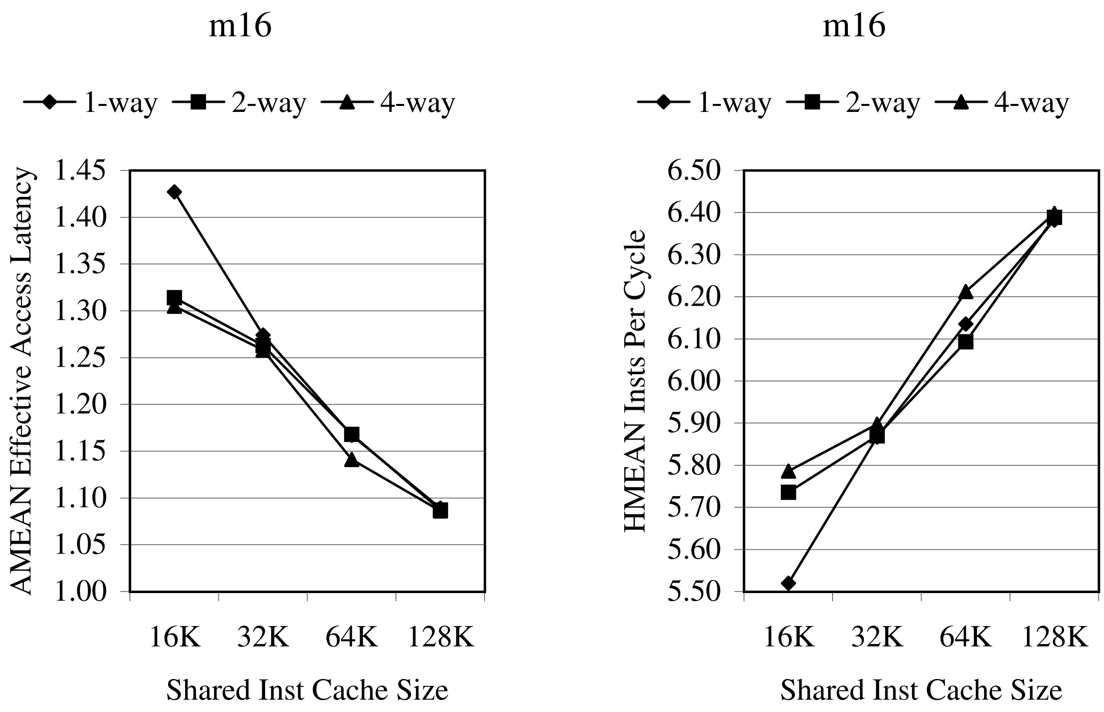


Figure 5.63: SPEC CFP95 effective access latency and performance with 1-way, 2-way, and 4-way associative shared instruction caches for 16 processing units.

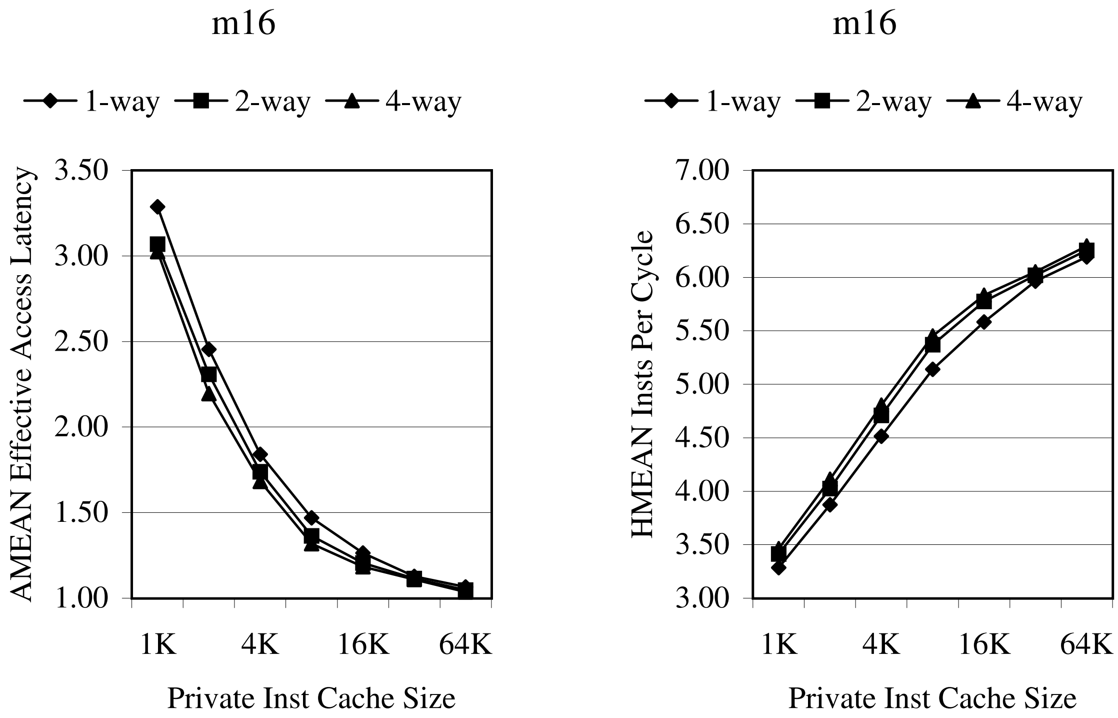


Figure 5.64: SPEC CFP95 effective access latency and performance with 1-way, 2-way, and 4-way associative private instruction caches for 16 processing units.

5.8.3.4 Real versus Ideal

In order to put the results of this instruction cache study in perspective, it is worthwhile to consider whether further improvements in instruction cache operation (aside from changing the hit latency) might provide further improvements in overall processor performance. An appropriate means of making such a determination is to compare the difference in overall processor performance using a real instruction cache (like the ones already studied) and using an ideal instruction cache. An ideal instruction cache has the same hit latency as a real instruction cache. However, unlike a real instruction cache which obviously has finite resources, it has infinite storage capacity (never has a miss), and it has infinite bandwidth (never has an address or a data conflict). This comparison presents results for both real shared instruction caches and real private instruction caches. (The ideal cache is the same in either case.)

Figure 5.65 compares the performance on the SPEC CINT95 benchmarks of real 2-way associative shared instruction caches with 16K-byte, 32K-byte, 64K-byte, and 128K-byte of banked storage to that of an ideal instruction cache for processor configurations of 2, 4, 8, or 16 processing units running a single task per processing unit (one-at-a-time). Just beneath this figure, Figure 5.66 provides the same comparison on the SPEC CFP95 benchmarks. Figure 5.67 compares the performance on the SPEC CINT95 benchmarks of real 2-way associative private instruction caches with 8K-byte, 16K-byte, 32K-byte, and 64K-byte of duplicated storage (for each processing unit) to that of an ideal instruction cache for processor configurations of 2, 4, 8, or 16 processing units running a single task per processing unit (one-at-a-time). Just beneath this figure, Figure 5.68 provides the same comparison on the SPEC CFP95 benchmarks.

Considering the shared instruction cache organizations, the integer benchmarks show small incremental performance increases up to the biggest real shared instruction cache used. These improvements diminish as the instruction cache size is doubled. Nevertheless, an ideal instruction cache offers a small but distinguishable performance advantage over even the biggest real instruction cache. This improvement, though, is due to the infinite bandwidth rather than the infinite storage aspect of the ideal instruction cache, as miss rates are low and conflicts are high, relatively speaking. The floating point benchmarks are similar to the integer benchmarks, but the performance differences between instruction cache sizes are smaller. Furthermore, there is no recognizable performance difference between the biggest real instruction cache and the ideal instruction cache.

Considering the private instruction cache organizations, the integer benchmarks show definite performance increases up to the biggest real private instruction cache used. These improvements diminish somewhat but still remain steady as the instruction cache size is doubled. Nevertheless, an ideal instruction cache offers a significant performance advantage over the biggest real private instruction cache considered. Clearly, this improvement is due to the infinite storage rather than the bandwidth aspect of the ideal instruction cache, as miss rates are high and conflict are non-existent. Again (as with the shared instruction cache), the floating point benchmarks are similar to the integer benchmarks, but the performance differences between instruction cache sizes are smaller. However, there is a recognizable performance difference between the biggest real instruction cache and the ideal instruction cache that is more apparent as more processing units are used.

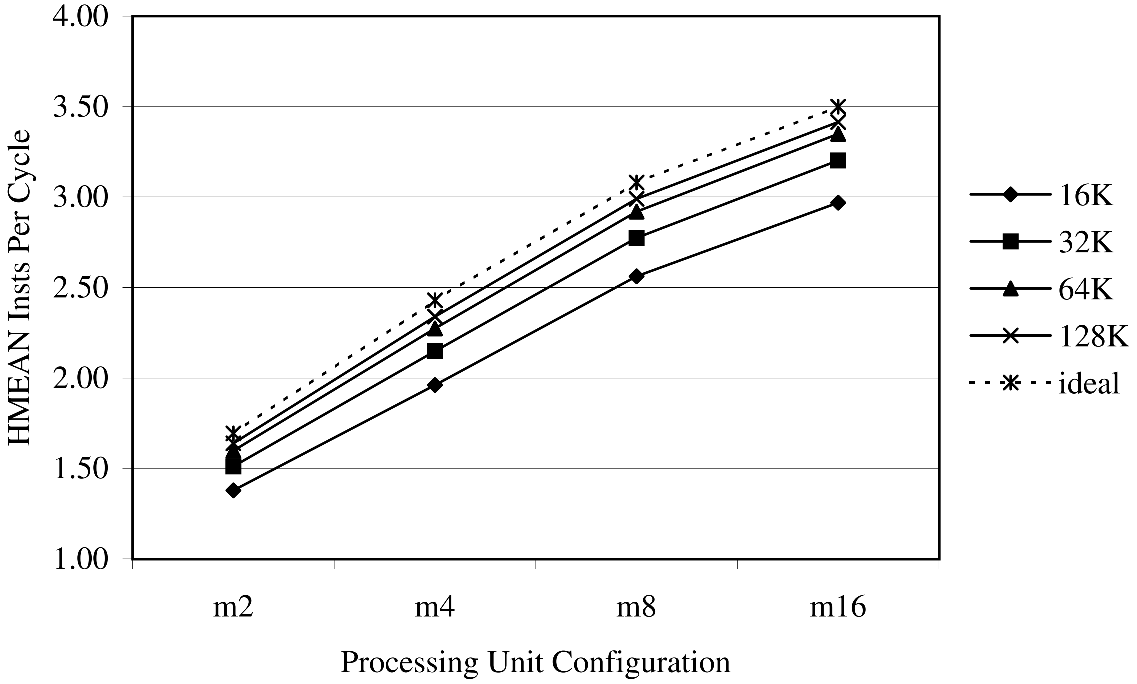


Figure 5.65: SPEC CINT95 performance of real 2-way associative shared compared to ideal instruction caches for 2, 4, 8, and 16 processing units.

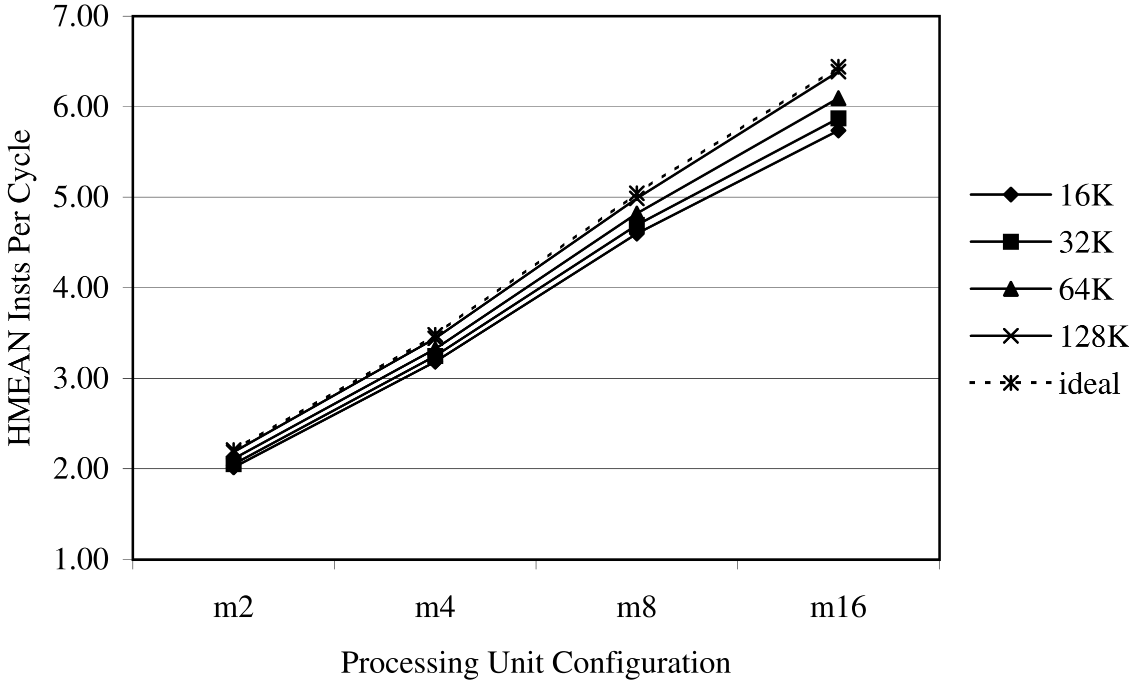


Figure 5.66: SPEC CFP95 performance of real 2-way associative shared compared to ideal instruction caches for 2, 4, 8, and 16 processing units.

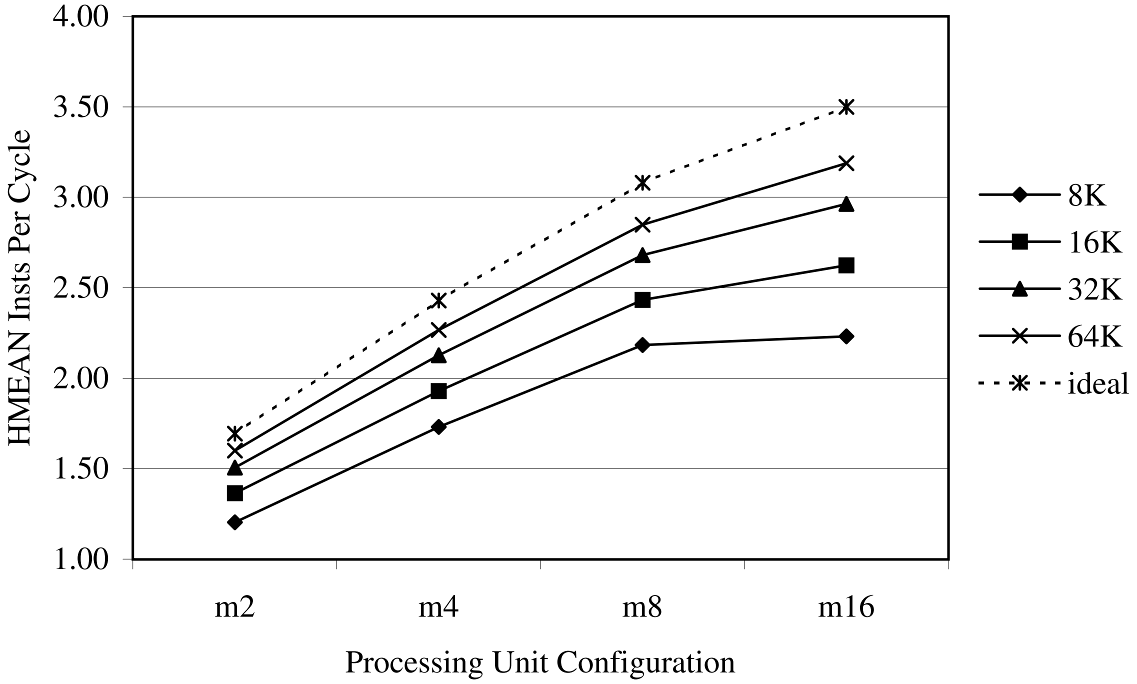


Figure 5.67: SPEC CINT95 performance of real 2-way associative private compared to ideal instruction caches for 2, 4, 8, and 16 processing units.

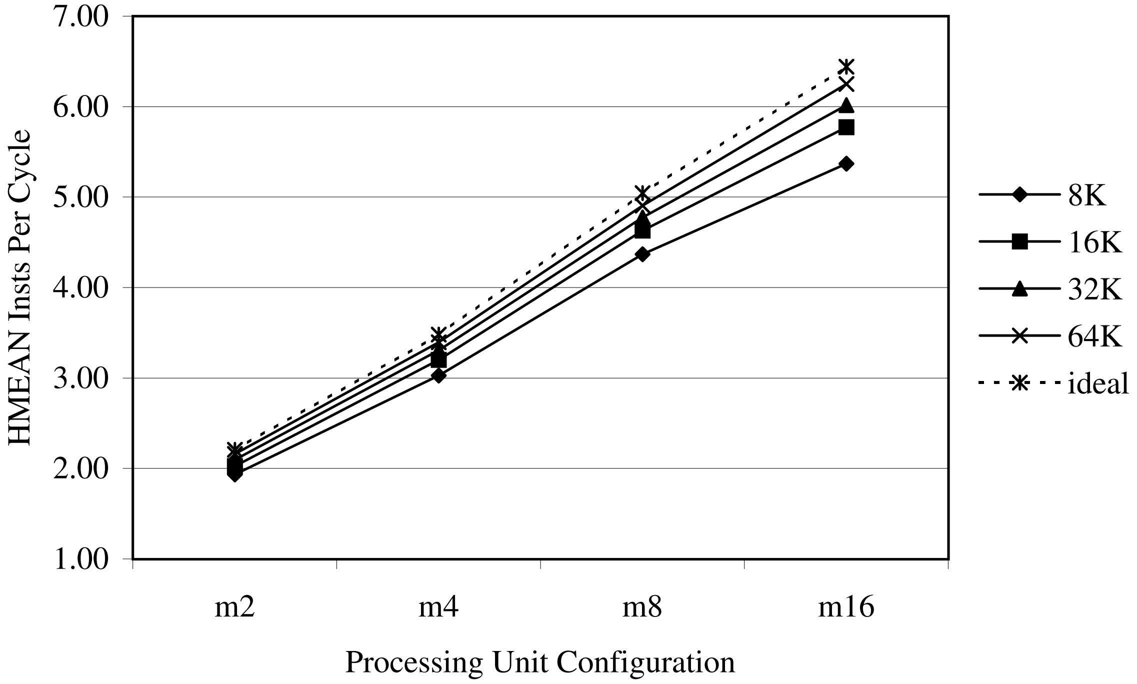


Figure 5.68: SPEC CFP95 performance of real 2-way associative private compared to ideal instruction caches for 2, 4, 8, and 16 processing units.

5.9 Summary

This chapter studied the instruction supply of a multiscalar processor. To do so, the concept of program sequencing, the cycle of instruction address generation and instruction memory access used to supply instructions, was discussed. In particular, the key differences between using a single program counter for sequencing (as conventional processors do) and multiple program counters (as multiscalar processors do) were detailed. With the abstract concepts described, the concrete requirements of multiscalar program sequencing were established in terms of control flow prediction and instruction memory accesses to motivate the design and evaluation of the two principal components of the instruction supply, the hierarchical predictor and the instruction memory.

5.9.1 Hierarchical Prediction

The hierarchical predictor was described as two aspects, inter-task and intra-task, of one prediction mechanism. The design of the hierarchical predictor demonstrated that both the inter-task and intra-task aspects could be provided using components based on existing prediction technology. To this end, the hierarchical predictor design was discussed in terms of using several well-known prediction schemes (address-based, path-based, global-pattern-based, and self-pattern-based) for both inter-task and intra-task predictors. The inter-task and intra-task predictors were evaluated in isolation of one another using each of these predictor designs to determine the best scheme for each. Using the best inter-task scheme (path-based) and the best intra-task scheme (global-pattern based), the impact of the hierarchical predictor on overall performance was evaluated with different amounts of prediction state devoted to each predictor.

In terms of hierarchical prediction, the experiments indicated that both inter-task prediction and intra-task prediction can be important performance factors. Moreover, the overall performance was usually much more sensitive to the inter-task than to the intra-task prediction accuracy. This behavior was expected, since the overall instruction window is not broken down in the case of intra-task mispredictions, but is in the case of inter-task mispredictions. Nevertheless, it is important to note that these conclusions primarily reflect the observed performance for SPEC CINT95 programs (with appreciable performance differences among both inter-task and intra-task predictors), rather than for the SPEC CFP95 programs (with no appreciable performance difference among either inter-task or intra-task predictors).

Among the predictors considered – path-based, global-pattern-based, self-pattern-based, and addressed-based – the path-based and global-pattern-based consistently performed best across the range of SPEC CINT95 and SPEC CFP95 programs for both inter-task and intra-task prediction. On the basis of these results as well as some considerations of design complexity, this work concluded that the best hierarchical prediction scheme is provided by combining a path-based inter-task predictor with a global-pattern-based intra-task predictor. In addition, for this combination, varying the amount of prediction state from 64K-entry to 8K-entry for each individual predictor showed that there is no appreciable difference in prediction accuracy among these sizes.

5.9.1.1 Benchmark Differences

In the case of the floating point programs, both intra-task and inter-task misprediction ratios were extremely low for the best predictors, arithmetic means in the range from 0.01 to 0.02 mispredictions per prediction, with little or no difference in terms of processor performance if a real predictor or a perfect predictor was used. The integer programs, on the other hand, had reasonably high misprediction ratios for even the best intra-task and inter-task predictors, arithmetic means in the range from 0.06 to 0.07 mispredictions per prediction, with a large overall performance difference between real and perfect predictors for inter-task predictors and a small one for intra-task predictors.

The SPEC CINT95 programs exhibited a range of moderate inter-task and intra-task misprediction rates among the prediction schemes studied. However, the inter-task and intra-task misprediction rates for the SPEC CFP95 programs were extremely low for all prediction schemes studied, regardless of their sophistication. As a result, for integer programs the overall performance behavior was relatively sensitive to the prediction scheme choice, while for floating point programs it was relatively insensitive. Therefore, caution ought to be exercised when considering these conclusions in the context of new applications.

5.9.1.2 Inter-Task Aspect

For the inter-task prediction, the path-based and global-pattern-based predictors had roughly half the misprediction rate of the self-pattern-based and the address-based predictors. As already described, though, these misprediction rate differences had no appreciable impact on the SPEC CFP95 benchmarks. In contrast, on the SPEC CINT95 benchmarks, these differences in misprediction rate, resulted in performance differences that grew as the number of tasks in concurrent execution.

The impact on overall performance was nearly imperceptible with 2 processing units, but it reduced performance by roughly 20% with 16 processing units. Still, even with the most accurate prediction schemes, performance was far below that which is achievable using a perfect inter-task prediction mechanism. Again, the performance differences grew as the number of processing units. For 2 processing units, the difference between perfect and the best prediction scheme was imperceptible, while for 16 processing units performance was degraded by roughly 30%.

5.9.1.3 Intra-Task Aspect

For the intra-task prediction, the path-based and global-pattern-based predictors again had the best misprediction rate across benchmarks. The self-pattern-based predictor started out with a comparable misprediction rate for 2 processing units, but steadily increased to more than 3 times the misprediction rate for 16 processing units. The address-based predictor provided nearly the same misprediction rate (except for the 2 processing unit configuration) on the SPEC CINT95 programs, but had roughly 2 times the misprediction rate on the SPEC CFP95 programs.

As for the inter-task aspect, these intra-task misprediction differences had no perceptible performance impact on the floating point programs. However, for the integer programs there was as much as a 12% performance difference between the path-based, address-based, and global-pattern-based predictors compared to the self-pattern-based predictors. Even so, the difference between the best intra-task predictors and a perfect intra-task predictor was quite small, only about 5%.

5.9.2 Instruction Memory

The instruction memory was likewise described as two parts. First, the task cache which provides access to task descriptors for inter-task sequencing was described. Second, the instruction which provides access to the actual instruction for intra-task sequencing was described. Both the task cache and instruction cache were evaluated in terms of miss rate, effective access latency, and overall impact on performance for a variety of sizes and associativities. Moreover, the instruction cache was evaluated using two different high-bandwidth organizations, interleaved shared cache and duplicated private cache, as well as a hybrid organization of these two. Overall, the results made it clear that processor performance is impacted primarily by the task cache and secondarily by the instruction cache, though inadequate attention to either may lead to severe performance degradations.

In terms of the instruction memory, the experiments indicated that both the task and instruction caches can be important performance factors. Yet, the overall performance was usually more sensitive to the task cache behavior than to the instruction cache behavior. This behavior was expected, since task cache accesses expand the overall instruction window, while instruction cache accesses only expand the individual instruction windows of their corresponding processing unit. The inter-task sequencing that is supported by the task cache is a serial process that when stalled affects the amount of instruction-level parallelism that is exposed to the processing units collectively. In contrast, the intra-task sequencing that is supported by the instruction cache is a parallel process that when stalled only affects the amount of instruction-level parallelism that is exposed to a processing unit individually.

5.9.2.1 Benchmark Differences

Comparing the floating point and the integer programs, there was significantly more instruction memory locality in the floating point than in the integer programs. That is, the floating point programs experienced almost no performance degradation across a range of task cache sizes ranging from 64 to 1024 entries and shared instruction cache sizes ranging from 8K-byte to 64K-byte. On the other hand, the integer programs experienced more significant, though still relatively limited, performance degradations across the same range of task cache and shared instruction cache sizes.

Since both types of programs performed well even with limited cache sizes it came as no surprise that using ideal task and instruction caches (ones that always hit with a fixed access latency and had unlimited bandwidth) did not provide an appreciable benefit in terms of overall performance. Though poor task cache and/or instruction cache behavior is sure to limit the ability of a multiscalar processor to extract instruction-level parallelism, there is little demand for more sophisticated solutions to handle instruction memory than those considered here for the applications studied, the SPEC CINT95 and CFP95 benchmarks.

5.9.2.2 Task Cache

Considering the task cache experiments, it was found that the task cache of a multiscalar processor exhibited typical cache-like behavior. That is, the miss rate of the task cache decreased steadily as the task cache size increased, leveling off once the working set of program task descriptors fit into the task cache. Likewise, the use of associativity (with diminishing returns as with the cache size) improved the miss rate of the task cache, with most of the improvement moving from 1-way associativity to 2-way associativity and only somewhat of an improvement moving from 2-way to 4-way associativity. In fact, the task caches considered here performed so well, that when an ideal task cache was used to determine whether further improvements in task cache operation (aside from changing the hit latency) might provide further improvements in overall processor performance, it was found that an ideal task cache offered no discernible performance improvement over a sizeable (around 1024 entries) real task cache.

5.9.2.3 Instruction Cache

Considering the instruction cache experiments, it was found that the instruction cache of a multiscalar processor, like the task cache, exhibited the same typical cache-like behavior with respect to decreases in miss rate as a result of increases cache size and associativity. Nonetheless, with regard to cache design in general, this work differs from most previous instruction cache studies since it focused on high-bandwidth, low-latency instruction cache design, and it assessed the impact of such an instruction cache design on the overall performance on an aggressive instruction-level parallel processor, specifically a multiscalar processor.

To this end, this study of the instruction cache for a multiscalar processor considered the design and evaluation of two more or less opposite approaches to the design of the instruction cache, a (banked) shared instruction cache for all processing units or a private instruction cache for each processing unit. These experiments indicated that, a (banked) shared instruction cache design with a multiple cycle access outperformed a private instruction cache design with a single cycle access latency unless each private instruction cache was comparable in size to the shared instruction cache. Yet, the cost in terms of cache storage (which is proportional to the number of processing units) required for a successful private cache design did not appear to make it favorable alternative to using much less cache storage for a successful (even if multiple cycle) shared cache design.

Nevertheless, these experiments showed that there were sizeable performance gains to be attained if the access latency of the shared instruction cache design was reduced. Of course, a successful design ought to be one which does not require cache storage that is proportional to the number of processing units. From the experiments with the private and shared approaches, this work proposed a hybrid design, based on different elements of each, that made up for the weaknesses of their respective approaches by combining their respective strengths. This hybrid design used small private caches to provide single cycle access, but connected them via a bus to a level of shared cache with multiple cycle access to overcome the performance loss associated with the high miss rate and high miss penalty that plagued the purely private cache approach.

The hybrid instruction cache design performed as well as a shared instruction cache design with 1 cycle hit latency for small numbers of processing units, but was significantly degraded for large numbers of processing units. In fact, the performance degradation was comparable to an increase of 1 cycle hit latency with a shared instruction cache for each increase of double the number of processing units. The reasons for this degradation were relatively straightforward. Because the private instruction caches had relatively high miss rate and the shared instruction cache had relatively low bandwidth, an increase in demand from more processing units could not be satisfied, and a decrease in performance resulted. Nevertheless, depending on the actual hit latency of a particular shared instruction cache design and the actual number of processing units which constituted the processor, a hybrid instruction cache design could provide better performance than a shared instruction cache design at comparable cost.

In order to determine whether further improvements in instruction cache operation (aside from changing the hit latency) would provide further improvements in overall processor performance, an ideal instruction cache was compared with real shared instruction caches and real private instruction caches. An ideal instruction cache offered a small but distinguishable performance advantage over the biggest real shared instruction cache. This improvement, though, was due to the infinite bandwidth rather than the infinite storage aspect of the ideal instruction cache, as miss rates were low and conflicts were high, relatively speaking. However, an ideal instruction cache offered a significant performance advantage over the biggest real private instruction caches considered. Clearly, this improvement was due to the infinite storage rather than the bandwidth aspect of the ideal instruction cache, as miss rates were high and conflicts were non-existent. Overall, though, it appears unnecessary to consider further improvements in instruction cache operation (with respect to the benchmarks studied) so long as the sizes considered in this work are practicable.

Chapter 6

Data Supply

The instruction and data processing in a multiscalar processor must be supported by appropriate instruction and data supplies, since effective extraction of instruction-level parallelism requires a plentiful amount of both. The previous chapter studied the instruction supply portion. This chapter studies the data supply of a multiscalar processor, with the first half concentrating on background and the second half on design and evaluation.

Section 6.1 gives an overview of the components, register file and data memory, used to supply register and memory values to a multiscalar processor. Section 6.2 explains the differences between register and memory namespaces that lead to different solutions to the data supply problem. Section 6.3 discusses the data supply requirements of a multiscalar processor. Section 6.4 reflects on concerns/challenges with respect to the design of the data supply components.

Section 6.5 investigates the design of the register file component of the data supply. Section 6.6 provides an evaluation of the register file designs. Section 6.7 investigates the design of the data memory component of the data supply. Section 6.8 provides an evaluation of the data memory designs. Section 6.9 gives a summary of this chapter.

6.1 Overview

The data supply for a multiscalar processor comprises a collection of register files and caches as well as specialized buffers and tables to support the processing units, as shown in Figure 6.1. In the case of the register files and caches, the individual components are similar to their counterparts in conventional processors. In the case of the specialized buffers and tables, the individual components are novel structures specifically developed for multiscalar processors. These structures may be adapted for use in conventional processors, but this aspect of their design is not addressed in this work.

6.1.1 Basics

Regardless of whether the individual components are similar to or different from those used in conventional processors, their organization (as studied here) is intended to support the multiscalar concept of dividing the dynamic instruction sequence into tasks and executing these tasks on multiple processing units in parallel. This parallel execution is a more challenging design problem for the data supply than for the instruction supply because data values are produced as well as consumed among the processing units. Hence, particular attention must be paid to ensure that accesses maintain the appearance of serial program semantics.

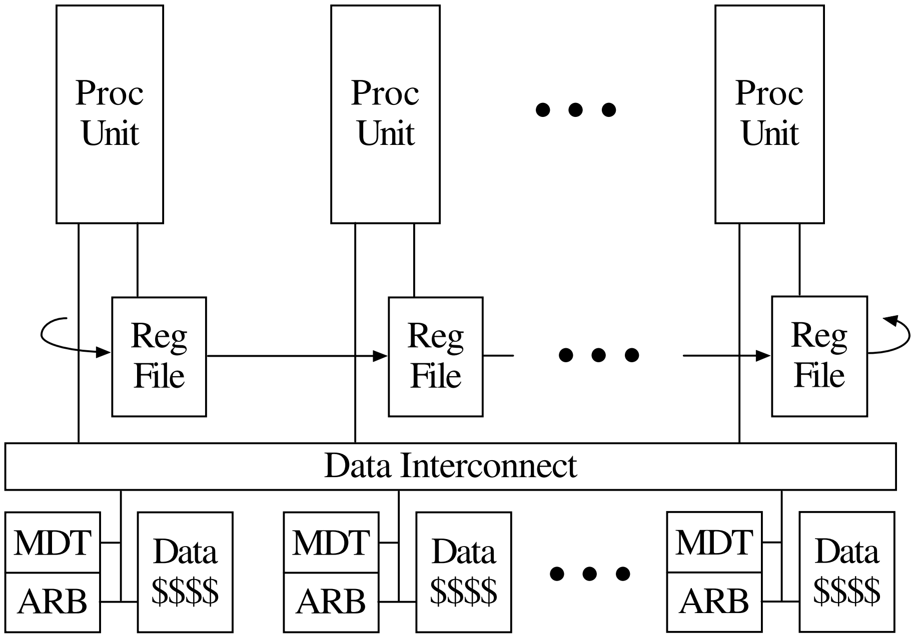


Figure 6.1: The organization of data supply components and the processing units in a multiscalar processor.

6.1.1.1 Register File Components

The register file components (collectively identified as **Reg File** in the figure) of the data supply consist of a collection of register files, one per processing unit, connected together via a point-to-point unidirectional communication ring. In addition, the register file of each processing unit consists of multiple distinct register files (separate past, present, and future register storage as needed) to support the demands of speculative execution among tasks. This decentralized organization is well-suited to multiscalar execution characteristics, since tasks provide an effective agent to exploit temporal locality in register accesses.

As with other multiscalar processor components, the operation of the register file may be divided into an intra-task aspect and an inter-task aspect. The intra-task aspect is straightforward. A processing unit reads/writes the register locations produced/consumed within a task in the same way as a conventional processor does. However, the handling of the inter-task aspect, where reads/writes of register locations might involve values produced/consumed by earlier/later tasks, is somewhat more involved.

Though each processing unit has its own individual private register file, the appearance of a collective shared register file must be maintained because the parallel execution of tasks corresponds to the dynamic instruction sequence of a serial program. In order to maintain the appearance of a collective shared register file among the individual private register files, the values produced by earlier tasks must be propagated to later tasks and consumed by them with the appearance of program order.

This problem is difficult to manage for the data memory part of the data supply, but is not so difficult for the register file part because of the applicability of compiler analysis. That is, the compiler is capable of completely specifying and orchestrating all register communication that occurs among tasks, thereby simplifying the design of this decentralized register file organization. The details of the design studied in this work will be described shortly.

6.1.1.2 Data Memory Components

The data memory components consist of a collection of caches (**Data $$$$** in the figure) as well as specialized buffers (address resolution buffer or **ARB** in the figure) and tables (memory dependence table or **MDT** in the figure) connected via an interconnect (here a crossbar) to the processing units. This decentralized cache organization coupled with specialized support to handle dependences as well as to provide storage (in the form of versions) for the demands of speculation among tasks is well-suited to multiscalar execution characteristics.

As with the register file components, the operation of the data memory may be divided into an intra-task aspect and an inter-task aspect. Likewise (for the same reasons as the register file), the intra-task aspect of the data memory is straightforward, and the inter-task aspect is somewhat more involved. However, unlike the register file components which are provided as private copies for each processing unit, the data memory components are shared among the processing units. This shared organization for the data memory is favored over the private organization for the register file because of the limitations of the compiler with respect to specifying and orchestrating memory communication among tasks in a manner comparable to the handling of register communication.

A brief description of the shared design studied in this work follows; the details will be described shortly. Though a private design similar to that of the register file was proposed by Franklin [25, 26], the description provided does not adequately address the concerns that must accompany a robust solution to the problem of cache-coherence in presence of speculative execution. An alternative private design, proposed by Gopal, Vijaykumar, Smith and Sohi [30], does address these concerns, but relies on a bus-based approach, similar to that used for multiprocessors, that may not accommodate many processing units and/or multiple tasks per processing unit (in its present form).

Due to the differences between the register and memory namespaces (identified below), it is problematic for the compiler to specify and orchestrate memory communication in the same way as register communication. As a result, memory accesses may be executed in a speculative manner that can lead to dependence violations. Such dependence violations do not occur for register accesses because the compiler disambiguates and synchronizes the producers and consumers of register values. To compensate for the limitations of static compiler techniques, the address resolution buffer [27] and the memory dependence table [57] may be used to perform dynamic disambiguation and synchronization functions.

The address resolution buffer tracks all memory accesses and disambiguates their addresses to ensure that the serial semantics of the program are maintained. Accesses that do not violate memory dependences during execution proceed without delay as intended. Accesses that do violate memory dependences are detected and all subsequent execution is squashed. (A selective means to perform a squash might be possible, but such a mechanism is not studied in this thesis.) The memory dependence table tracks memory dependence violations that occur in this way during execution. Using this information, the memory dependence table then synchronizes subsequent executions of these producer and consumer instructions to avoid dependence violations.

6.1.2 Insights

The combined behavior of the register file and the data memory mechanisms determines the characteristics of communication between producers and consumers of data values and the performance that may be obtained in terms of the data supply. The architectural register and memory storage of a multiscalar processor appear as single logical register and memory namespaces to running programs, regardless of their actual physical design. This condition exists because it is a fundamental concept underlying the serial program model that multiscalar processors support.

Yet due to the characteristics of multiscalar execution, any design that supports this condition is complicated by the fact that it must provide high-bandwidth and low-latency access to register and memory storage to sustain speculative execution across multiple processing units. In this context, a successful design of the data supply is one that provides effective and efficient, register and memory access mechanisms that address the fundamental issues (described in detail later in this chapter) common to both namespaces: storage, communication, speculation, and synchronization.

Despite these common fundamental issues, the inherent differences in the register and memory namespaces (also described in detail later in this chapter) give rise to different solutions to the data supply problem. That is, to accommodate the inherent differences in the two namespaces, the data supply studied in this work comprises two distinct design solutions, one for the register namespace and another for the memory namespace. The register storage is managed statically, entirely by the compiler. The memory storage is managed dynamically, entirely by the hardware. Together these designs provide a comprehensive data supply solution for a multiscalar processor.

6.1.2.1 Register File

The design and evaluation of the register file performed in this chapter demonstrates that a decentralized organization that places the complexity in software rather than hardware can provide an effective and efficient mechanism for high-bandwidth and low-latency, speculative register access in a multiscalar processor. The results of this study support this assertion in two step approach. First, the evaluation considers three behavior aspects of the register file design: (1) the register traffic per useful task and the register traffic per useful instruction, (2) the total register traffic and the link register traffic, and (3) the register task travel distance. Second, the evaluation investigates the impact on overall performance for changes to the register file bandwidth and/or register file latency characteristics. Both parts of the evaluation are performed using a full range of processing unit configurations for a multiscalar processor.

For the behavior oriented register file experiments, the results of this study indicate that register traffic is held in check via the filtering that naturally occurs as new register values are created and old register values are killed, both with and without the effect of mis-speculation. Along the same lines, the results indicate that the register task travel distance is held in check, due to the same filtering effect, as the number of tasks in concurrent execution grow, even taking into account the effects of mis-speculation. Moreover, the experimental evidence from this study shows that though the total register traffic per cycle (with mis-speculation as well as without mis-speculation) increases steadily as the number of tasks in concurrent execution increases, the link register traffic per cycle actually decreases as the number of tasks increases. Overall, from the combination of these behaviors, it may be concluded that sufficient register communication locality exists among tasks to support an effective and efficient decentralized register file design in a multiscalar processor.

For the performance oriented register file experiments, the results of this study indicate that the register file latency rather than the register file bandwidth is the critical factor. In particular, the performance impact of register file latency is that it reduces the multiscalar processor performance of more aggressive processing unit configurations to the performance of less aggressive processing unit configurations. In addition, this performance impact is more significant for the SPEC CINT95 than for the SPEC CFP95 benchmarks. The performance impact of register file bandwidth is similar to that of register file latency. However, the magnitude of this impact for register file bandwidth is relatively insignificant compared that of register file latency according to the results of this study. To be more specific, the processor performance improves only a small amount and at low rate that levels off rapidly as the register bandwidth is increased. In contrast, the processor performance improves a large amount and at a high rate that remains steady as the register latency is decreased.

6.1.2.2 Data Memory

The design and evaluation of the data memory performed in this chapter demonstrates that a decentralized organization that relies solely on hardware can provide an effective and efficient mechanism for high-bandwidth and low-latency, speculative memory access in a multiscalar processor. In particular, by combining two specialized components, the address resolution buffer (ARB) and the memory dependence table (MDT) with a more or less conventional (yet aggressive in terms of its overall bandwidth and latency characteristics), it is possible to deliver a design capable of supporting multiscalar execution. Of particular significance, this study indicates that though the characteristics of the conventional components, the data caches, are responsible for setting the level of performance that can be provided by the data memory, the characteristics of the specialized components, the address resolution buffer and the memory dependence table, are responsible for determining how close to this level the actual performance of a multiscalar processor can come (even for ideal data caches).

For the data cache experiments, the results indicate that (like the task and instruction caches of the previous chapter) the data cache exhibits typical cache-like behavior (with the same characteristics as well). Even so, the load latency characteristics of the data cache are much different among the SPEC CINT95 and SPEC CFP95 benchmarks relative to different processing unit configurations as well as in absolute terms. Though the effective load latency is relatively constant across processing unit configurations for the SPEC CINT95 benchmarks, it increases by as much as a cycle each time the number of processing units is doubled for the SPEC CFP95 benchmarks. In spite of this behavior, a high level performance is sustained for SPEC CFP95 programs, in contrast to the SPEC CINT95 programs which are quite sensitive to the load latency. The reason for the difference in observed performance in connection with that same type of data cache behavior is that higher levels of parallelism in the floating point benchmarks affords greater latency tolerance than in the integer benchmarks. To be specific, this study finds that performance declines more rapidly for the integer benchmarks (about 12% per cycle) than for the floating point benchmarks (about 3% per cycle) when the load hit latency increases. Exploring further, using an ideal data cache in place of a real data cache in this study indicates that there is need for further consideration of the bandwidth and latency aspects of data cache designs in a multiscalar processors. In particular, the results of this study show that an ideal data cache often provides significant improvement in terms of overall performance. This improvement is more pronounced for the floating point than for the integer benchmarks. Yet, the results of this work indicate that this effect is due to the infinite bandwidth rather than the infinite storage characteristics of the ideal data cache considered.

For the address resolution buffer experiments, the results show it to be a critical performance factor that can reduce performance significantly – as a result of processor stall cycles – in the event it overflows and cannot track additional loads and stores. The causes for overflow identified in the studies of this thesis are insufficient number of entries, insufficient associativity among the entries, and/or insufficient version storage per entry. Of these three causes, the studies considered here point to the amount of version storage per entry as the actual root cause for the likelihood of address resolution buffer overflows. Regardless of the amount of version storage per entry (word or cache block), increasing the number of entries or the associativity among the entries results in steadily decreasing overflows and steadily increasing overall performance. Nevertheless, the relationship between the amount of entries and/or associativity required to avoid overflow and the amount of version storage per entry is an inverse one. That is, using a smaller version requires a larger number of entries and/or associativity to provide comparable performance to using a larger version. According to findings in this study, the use of a version that is consistent with the data cache block size appears to be the proper choice since it allows the address resolution buffer to better capture the same spatial and temporal locality as the data cache itself. Nonetheless, the results in this study also find that the SPEC CFP95 benchmarks require higher numbers of entries as well as higher associativity to avoid overflows than do the SPEC CINT95 benchmarks. Thus, it is important to remember that spatial and temporal locality are dependent on the working set and access pattern of the application as well as the nature of the instruction window, so an address resolution buffer design that suffices in one instance may not in another.

Likewise, for the memory dependence table experiments, the results show it to be a critical performance factor that can reduce performance significantly – as a result of processor squash (rather than stall) cycles – in the event it does not synchronize the blind speculative execution of loads and stores. In particular, the results contained in this thesis show that both the SPEC CINT95 and SPEC CFP95 benchmarks rely upon the dynamic synchronization of producers and consumers provided by the memory dependence table to obtain their highest levels of overall performance. Though the performance of the integer benchmarks is more sensitive to the synchronization provided by the memory dependence table than is the performance of the floating point benchmarks, the role of the memory dependence table increases in significance as the size of the task window on the dynamic instruction sequence increases for both. The experiments conducted in this study make this point clear by focusing on the performance impact with or without the use of the memory dependence table, instead of the specifics of its organization, for a range of processor configurations. An encapsulation of the key insights for this study is given by the following results. For both the SPEC CINT95 and SPEC CFP95 benchmarks, there is no appreciable difference in performance with or without the memory dependence table for small, less aggressive, multiscalar processor configurations (such as for 2 processing units or 2 tasks). However, there is a significant performance difference for large, more aggressive, multiscalar processor configurations (such as for 16 processing units or 16 tasks). Moreover, the relative performance difference with and without the memory dependence table is twice as much for the SPEC CINT95 as for the SPEC CFP95 benchmarks, while the absolute performance difference is roughly the same. To be specific, using a configuration of 16 processing units, the SPEC CINT95 benchmarks improve by nearly 1.00 instructions per cycle (absolute) or 40% (relative), and the SPEC CFP95 benchmarks improve by a little over 1.00 instructions per cycle (absolute) or 20% (relative).

6.2 Register versus Memory Namespaces

With the overview of the data supply just presented in mind, it is worth switching gears to consider why two different approaches, one for the register namespace and one for the memory namespace, were used to solve what is ostensibly the same data supply problem.

As instructions in tasks execute, data values are produced and consumed within the same task (intra-task communication) as well as among tasks (inter-task communication). In the load-store architectures that dominate modern instruction sets, these data values are bound to storage locations addressed via two distinct namespaces: the register namespace and the memory namespace. The basic differences between these two namespaces are the number of storage locations and the means by which these storage locations are specified. (Differences in the speed of register versus memory access are not relevant to this discussion.)

Register storage locations are limited in number (32 integer and 32 floating point registers in many machines). Memory storage locations are relatively unlimited in number (2 Giga-byte of addressable memory storage is common in many machines). As might be expected, this disparity in namespace size leads to differences in the way storage locations are specified. Register locations are specified explicitly in instructions using a small number of bits (typically 5 or 6). Memory locations are specified implicitly via an address computation performed as part of instruction execution providing a large number of bits (typically 32 or 64).

In terms of the design of the data supply, these basic namespace differences are important, but not as much so as their interaction with the usage of the namespaces. Register usage patterns may be analyzed by the compiler because (in most cases) the compiler chooses the binding between program variables and register storage. Memory usage patterns often may not be analyzed by the compiler since the programmer can exercise control over the binding of program variables and memory storage. In particular, the use of pointers in high-level languages, such as C, often stymies compiler analysis of memory usage patterns (more so for heap storage and less so for stack and global storage).

Yet, even if the compiler can analyze memory usage patterns, there is still the difficulty of static specification of memory storage locations. This situation is problematic for a multiscalar processor because data dependences need to be given in advance of task execution (as discussed in Chapter 2). In terms of efficiency, many register storage locations may be specified with a small number of bits or bit mask, while only a few memory locations may be specified with a comparable number of bits. Moreover, because memory locations are specified via a dynamic address calculation, a single static instruction may access multiple memory locations. This dynamic aspect of memory usage patterns makes their static specification especially difficult.

To accommodate what seem to be inherent differences in the two namespaces, the multiscalar processor studied in this work uses two distinct design solutions, one for the register namespace and another for the memory namespace, to address the data supply problem. The register storage is managed statically, entirely by the compiler. The memory storage is managed dynamically, entirely by the processor. Though it may be possible to use some combination of static and dynamic management for both register and memory storage, such designs are not considered further in this work. Moreover, using a purely static approach or a purely dynamic approach has a conceptual simplicity that may foster a better design.

6.3 Multiscalar Requirements

For a multiscalar processor, the architectural register and memory storage must appear as single logical register and memory namespaces to running programs, even though the actual physical design may not match this appearance. Because this behavior is a fundamental requirement of the serial program model that multiscalar processors support, it is a fundamental requirement of the data supply for a multiscalar processor. However, any design that supports this requirement is complicated by the fact that it must provide high bandwidth and low latency access to register and memory storage to sustain speculative execution across multiple processing units. Therefore, the data supply must be designed to provide effective and efficient, register and memory access mechanisms that address the key issues of storage, communication, speculation, and synchronization.

Though differences between the register and memory namespaces elicit different solutions to the data supply problem, the designs for each type of storage must still address these same issues. The storage mechanism must provide a means to bind data values to storage locations and to distinguish between multiple values that are accessed via the same name. The communication mechanism must satisfy true data dependences between producers and consumers in a correct and timely manner. The speculation mechanism must identify data dependence violations and maintain the consistency of the architectural and speculative states of the machine. The synchronization mechanism must coordinate the concurrent production and consumption of values to avoid performance degradations due to data dependence violations.

6.3.1 Storage

The storage mechanism is the basis of the data supply in that it provides the space to hold the data values. The characteristics of the space provided dictate to what extent the data supply allows the processor to expose and to exploit instruction-level parallelism. In particular, the storage mechanism must provide both architectural and speculative storage so that the precise state of the machine may be preserved in the course of speculative execution. In addition, the storage mechanism must provide storage to support multiple versions of a storage location. These versions are needed to serve as the means to rename register and memory storage, so that only true dependences guide parallel execution. Otherwise, anti-dependences (write-after-read or WAR) and output dependences (write-after-write or WAW) may unnecessarily limit the extent to which parallel execution may be achieved. The decisions of how many versions of a storage location to maintain and how to provide and manage the speculative and architectural storage are complex questions linked to the number and characteristics of the processing units and tasks. These issues are addressed later in the description of the actual designs studied.

6.3.2 Communication

Though the storage mechanism is the basis of the data supply, the communication mechanism is the agent through which the data is actually supplied to the processor. The communication mechanism must satisfy the true data dependences between producers and consumers of values. However, because many producers and consumers execute in parallel, attention must be given to ensure that values supplied are consistent with serial program semantics. The appearance that must be preserved in relation to the dynamic instruction sequence of the program for instructions accessing the same storage location may be described as follows: (i) the value a consuming instruction reads must be the value the nearest preceding producing instruction writes, and (ii) the value a producing instruction writes must persist up to the point the nearest succeeding producing instruction writes. In this context, nearest means the closest instruction, forward or backward in the dynamic instruction sequence in relation to the instruction being considered, that accesses the same storage location. In addition to maintaining this appearance for the sake of correctness, the communication mechanism must also deliver the values from producers to consumers in a timely manner for the sake of performance. That is, the communication bandwidth and latency characteristics must be such that the communication of data values between producers and consumers does not unduly impact performance.

6.3.3 Speculation

The catalyst that provides the opportunities for exposing and exploiting instruction-level parallelism via the storage and communication mechanisms is speculative execution. When dependences are speculated, whether control and/or data, the communication that occurs between producers and consumers – the values that producers write to storage and that consumers read from storage – must be treated as speculative. In the case of correct speculation, it is necessary to distinguish between speculative and architectural values for the purposes of precise updates from speculative to architectural machine state as well as to supply the values that preserve the serial semantics of the program. In the case of incorrect speculation, it is necessary to determine what values constitute the precise state of the machine and to discard those that are not needed. These requirements are fulfilled by using a speculation mechanism that works in conjunction with the storage mechanism and the communication mechanism. To be specific, the role of the speculation mechanism is (i) to identify dependence violations, (ii) to recover the proper architectural and speculative states when dependence violations occur (incorrect speculation), and (iii) to update the architectural state from the speculative state when dependences are resolved (correct speculation).

6.3.4 Synchronization

The use of speculative execution serves to improve the performance of the processor in general. However, in specific cases, namely when speculative execution results in dependence violations, the use of speculative execution may actually degrade the performance of the processor. To avoid the performance loss associated with dependence violations, a means must be provided to throttle the degree of speculative execution (at least in cases where there are dependences between producers and consumers and where the execution of these producers and consumers violates these dependences). An effective means of avoiding the dependence violations associated with speculation is to provide a synchronization mechanism for ordering the execution of producers and consumers. To be specific, the role of the synchronization mechanism is (i) to identify dependent producer and consumer instructions and (ii) to delay the execution of consumer instructions until such time as the data from the execution of producer instructions is available. However, it is important that the synchronization mechanism not delay the execution of instructions any longer than is necessary to ensure their proper ordering. Otherwise, performance may be degraded to a greater extent than if the dependence violations were allowed to occur.

6.4 Concerns/Challenges

The combined behavior of the register file and the data memory mechanisms determine the characteristics of communication between producers and consumers of data values and the performance that may be obtained in terms of the data supply. Yet, the inherent differences in the register and memory namespaces (already described) lead to different solutions to the data supply problem. Likewise, because the approaches to the register file and the data memory are different, there are different concerns/challenges with respect to their design to ensure that overall the data supply functions effectively and efficiently.

6.4.1 Register File

With respect to the register file, the concern is not the bandwidth and latency between each processing unit and its register file, but instead it is the bandwidth and latency between adjacent register files. Between each processing unit and its register file there is sufficient bandwidth to accommodate the needs of the processing unit. Moreover, between each processing unit and register file, register read and write accesses occur in the same cycle as initiated similar to the characteristics of conventional processors. Nevertheless, because the dynamic instruction sequence is divided among the processing units, the bandwidth and latency between the register files can have a critical impact on performance since inter-task register file communication must occur between the individual register files.

The two primary issues for the design of the register file are correctness and performance. The correctness may be ensured so long as the software identifies last updates for registers and so long as the hardware propagates these updates among the register files of the processing units as directed and in the presence of speculative execution. The performance is not quite as straightforward for two reasons. First, to some extent the performance is dictated by the characteristics of the tasks that execute on the processing units. If a task consumes a register early that another task produces late, then there is little the register file can do to improve this situation. Collectively however, what the register file can do is limit the overall delay from when a register value is produced in one task to when this register register value may be consumed in another task.

6.4.2 Data Memory

With respect to the data memory, there is obviously concern for the bandwidth and latency of access. The use of cache memory is the best known method to increase bandwidth and/or decrease latency. So long as spatial and/or temporal locality exists in the data memory access characteristics of the processing units, caches represent an effective technique for this purpose. Yet, there is more to the handling of the data memory than just providing suitable bandwidth and latency. Unlike the register file in which communication is orchestrated by the compiler via static analysis, the communication in the data memory must be orchestrated by the hardware via dynamic mechanisms.

6.4.2.1 Conventional Components

The data cache which provides the architectural storage of the data memory is straightforward. Its design may be similar to the data caches that support conventional processors. Nevertheless, because the degree of speculation and instruction-level parallel execution may far exceed that of conventional processors, there must be concern to provide enough bandwidth to satisfy the processing units, and latency must be considered with respect to how much it affects the overall processor performance. Moreover, because processing units execute tasks concurrently, the data cache must provide the ability to handle many outstanding misses. The capability of performing other independent work along with handling misses is practically a requirement if a high-performance processor is to extract significant amounts of instruction-level parallelism [23].

6.4.2.2 Specialized Components

Though the characteristics of the conventional components of the data memory are fairly well understood, the specialized components (created to solve problems that had not been faced by conventional processors at the time multiscalar processors were proposed) are not. Thus, there is concern for whether or how well the address resolution buffer and the memory dependence table can fulfill their function in the data memory. In particular, the address resolution buffer needs to track the load and store accesses performed by the processing units. There is concern for how many addresses the address resolution buffer must be able to track in order to sustain the speculative execution of the processor. As for the memory dependence table, there is concern for how well it can actually identify producer and consumer instructions and synchronize their execution to avoid costly dependence violations.

The handling of memory accesses by the specialized components may be divided into intra-task (within a processing unit) and inter-task (among processing units) aspects. Though both aspects are of concern to insure correct execution, the focus in this study of the data memory is the inter-task rather than the intra-task aspect. Within a task, the correct ordering of loads and stores may be maintained by one of several of existing techniques for this purpose [67, 82]. As a result of using these techniques, there is no data speculative execution of loads with respect to stores within tasks since the addresses of all earlier stores must be resolved before a later load is allowed to execute. In this way, ensuring the correct behavior of memory accesses may be performed in a decoupled manner to simplify the problem as well as focus on the unique issues associated with multiscalar processors; in particular, the speculative execution of loads and stores among different tasks.

6.5 Register File Design

The register file of a multiscalar processor must provide the appearance of a single centralized register file because these are the semantics of the serial programming model. Nevertheless, to provide an efficient implementation, this work considers a register file design which actually consists of multiple decentralized register files. As a result of this organization, the operation of the register file is somewhat different from what is normally assumed. In a conventional processor, it is usually the case that a register value produced in one cycle may be consumed in the next cycle. Indeed, in terms of intra-task register communication this behavior is the same for a multiscalar processor as it is for a conventional processor. However, in terms of inter-task register communication, this behavior is not as simple to provide because each processing unit is associated with its own distinct register file.

Yet, if inter-task register communication were to be provided with a single shared register file, the demand for high register access bandwidth and low register access latency that drives instruction-level parallel execution may be difficult to provide with such an organization. In the case of register read accesses, the area required to build read ports is proportional to the square of the number of ports. In the case of register write accesses, the ability to support many write ports may adversely affect the delay incurred when write operations are performed. Yet, some register files of this kind (though not necessarily supporting a single logical register namespace) have been proposed and/or designed, *e.g.* the register files of the Cydra5 [71], the SIMP [58], the Warp [5], and the XIMD [101] processors. Nonetheless, this approach does not seem to be a good long-term solution, since alternatives that take advantage of temporal locality in register usage have been shown to be effective [28].

Though a decentralized register file design may seem more involved than a centralized register file design, there are really only two basic aspects for this approach to the register file design that require particular attention. First, in a decentralized design, the register values are produced on different processing units, and thus stored in different register files. As a result, not all register values are immediately accessible by instructions that execute on all processing units. Therefore, register values must be propagated to the other register files in order to make them available to instructions that execute on other processing units. Second, because register values are propagated from one processing unit to the next, a register value might not be available for consumption the cycle immediately following its production. The extent to which these factors affect the complexity of a decentralized register file is dependent upon the characteristics of its design.

Though there are a variety of approaches to go about solving this problem, the register file design for the multiscalar processor studied in this work, uses a combination of software and hardware to provide a register file that functions effectively and efficiently, placing the complexity in software rather than hardware. The register file consists of a collection of relatively simple register files, duplicated with one for each processing unit and connected via a point-to-point unidirectional communication ring. Each processing unit accesses its register file as a conventional processor does. However, communication between register files is orchestrated by the compiler and carried out by the processor to ensure that the register files remain consistent with respect to the serial semantics assumed by the programs that execute.

6.5.1 Software Role

The software, *i.e.* the compiler, plays a key role in the register file in terms of reducing the complexity of its design. That is, the compiler assists the hardware by providing information that would otherwise not be available (delaying register file actions) or otherwise have to be computed at run-time (complicating register file design). In particular, the hardware needs to know the registers that may be modified by a task and that must be propagated to other tasks, as well as which instruction of the task performs the last update of the register. Fortunately, all of this information is readily available from a simple analysis of the control flow graph and may be provided in a straightforward manner. (The basics and terminology used in the following description was presented in Chapter 2.)

Using static analysis of the program control flow graph, the compiler can identify which registers a task may modify, indicating them on the *create mask*. In addition, by combining this analysis of the control flow graph with analysis of the data flow graph, the compiler can identify which register values need not be propagated to other tasks, indicating them on the *kill mask*. As tasks are assigned to processing units, the create and kill masks may be used to determine (for each task) what registers may be modified by earlier tasks. A later task uses this information to place reservations on the indicated registers, so that if an instruction that accesses these registers executes, the instruction is made to wait for the value(s) to be propagated from the earlier tasks.

In addition to indicating which registers a task may modify, the compiler also may identify the last instruction in a task to update a register. The last updates of registers are identified by associating an additional bit field, a *forward bit* with every instruction in the static code that indicates whether the instruction ought to forward to other tasks its modification of a register as the last update of the register. Because a task may contain arbitrary control flow, the create mask must be conservative. In some cases it may not be possible to uniquely identify the last update, while in other cases no instruction may actually modify a register. To deal with both of these situations, the compiler inserts *release* instructions to release the value for a register that can no longer be modified by the execution of the task.

6.5.1.1 Handling Simple Task Control Flow

If the path of control in a task is simple (contains no conditional control flow), the compiler analysis to be performed is straightforward. The create mask is determined by noting the destination registers of all instructions within the task. The kill mask is an optimized version of the create mask that indicates those register values that are dead outside of the task. With only a single path of control, it is simple to determine which is the last instruction in a task to update a register. This instruction is tagged with the forward bit so that when the instruction is executed, the value created for its destination register is propagated to other processing units.

6.5.1.2 Handling Complex Task Control Flow

If the path of control in a task is complex (contains conditional control flow), the situation becomes more involved. The instructions may be tagged with forward bits as before, but the determination of the create mask may be somewhat problematic. As the dynamic path through the task is unknown, the create mask must allow for all possibilities and must reflect the union of register values created on all possible paths through the task. Depending upon the actual dynamic path of execution through the task, some register values indicated in the create mask may never be produced (due to the conservative nature of the create mask).

Unfortunately, it is not possible to determine which registers fall into this category until the end of the task is reached at run-time. As the registers on the create mask imply the need for synchronization with successor tasks, waiting until the end of the task to fulfill the obligations on the reserved registers can significantly delay the execution of successor tasks [96]. The performance penalty caused by the conservative nature of the create mask may be overcome with the use of a release instruction (as mentioned earlier). The function of this instruction is to release the reservations made on registers by the conservative create mask. The use of such release instructions is assumed for the register file studied in this work.

The release instructions are inserted at appropriate points in the code where it is known that a particular register value cannot be created by any subsequent execution of the task (because execution has proceeded down a path that cannot create the register value), even though the create mask indicates that the task might create the value. The execution of the release instruction causes the propagation of the specified register values. Thereby, successor tasks which might be waiting on the released register value are able to proceed without further delay.

A task that contains a procedure call requires special handling. The compiler may conservatively assume that all registers are created, causing an increase in the register traffic. Alternatively, the compiler may compute the exact create mask, requiring inter-procedural dependence analysis. Fortunately, most compilers (including the one used in this work) follow caller/callee save conventions for procedure calls. This practice motivates a simple, yet effective, solution. The register values that are created inside the procedure are not live after the procedure call returns. The only exception to this rule are values returned in a register and global variables allocated to a register. The create mask must include any values returned and any global variables modified by the callee procedure.

6.5.1.3 Examples

Figure 6.2 shows three examples of forwards and releases. In the figure, only the last update of a value down a path is a *forward*, other updates are indicated as *define*. The *release* indicates the affected values. All the variables are assumed to be register allocated.

The first example shows forwards and releases in the presence of acyclic control flow in the task. If *cond1* is true, then x and y are forwarded by the instructions that compute them. Since z is not defined down this path, it is released. The same actions are taken if *cond2* is true. Note that the earlier define of x (if *cond1* is false) does not forward it. If *cond2* is false, then the instruction that computes z forwards it, and x and y are released. Since w is independent of control flow, it is always forwarded.

The second example shows a task which contains an entire loop. All iterations of the loop are performed within the task. The registers that are created in the loop can only be forwarded/released when the loop is exited. This behavior is manifest because a guarantee of no more defines of the registers can be given only at the exit points of the loop. In the example, there are two exit points from the loop, the loop condition ($c \,||\, !z$) being found false or the break being executed.

All of x, y, z and c are defined in the loop. If the loop is exited via the break, then z is forwarded, and x, y and c are released. If the loop is exited due to the fall through of the loop, then all are released. Note that since the two exit paths merge, the hardware may encounter releases for the registers, even though it had already forwarded/released them in the break. In this case, the redundant releases are ignored.

The third example shows a task which contains a procedure call. That is, all the instructions of *foo* are performed within the task. Assuming no global register allocation in the procedure *foo*, the return value, z, is released after the call returns. All the registers created within the procedure *foo* are completely “hidden” from other tasks because they are neither included in the create mask of the task nor forwarded/released from within *foo*.

```
if (cond1) {
    x = valxif;
    y = valycond1;
}
else {
    x = valxelse;
    if (cond2) {
        x = valx;
        y = valycond2;
    }
    else {
        z = valz;
    }
}
w = valw;
```

```
while (c || ! z) {
    if (x < y) {
        z = x;
    }
    else if (y < x) {
        z = y;
    }
    else {
        z = 0;
        break;
    }
    x = 2 * y – z;
    y = 2 * x + z;
    if (! (x && y)) {
        c = 0;
    }
}
```

```
x = max(a, b);
y = min(c, d);
if (cond) {
    z = foo(x, y);
}
w= x + y + z;
```

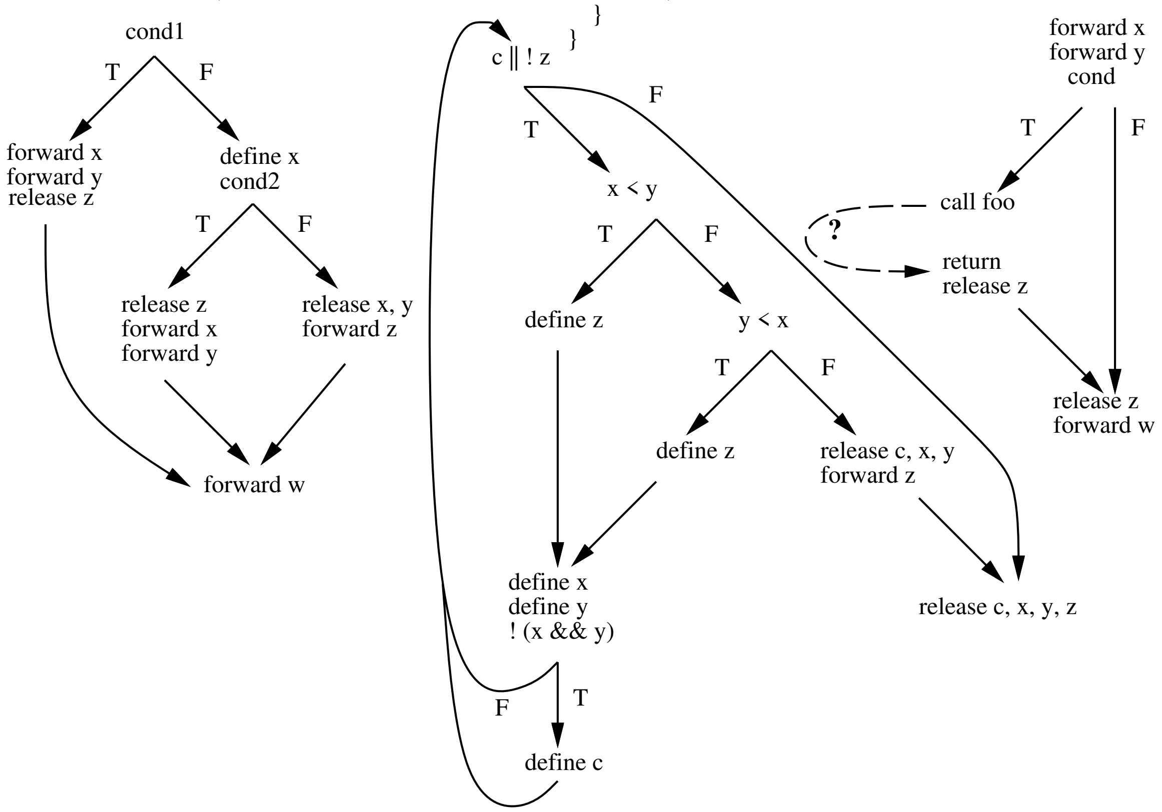


Figure 6.2: Examples using forward and release for tasks with simple control flow, loop, and procedure call.

6.5.2 Hardware Role

Even though the compiler provides the information to direct register communication, it still relies on the hardware to enact register communication. The basic components of the register file in a multiscalar processor are similar to those of a conventional processor, except for the use of a modified register storage and the addition of register control bit masks (see Figure 6.3). Since these changes to the register file (explained below) are modest, the overall design ought to be comparable to that of a conventional register file, and there ought to be no significant impact on the critical timing paths of register access.

The register storage serves as a repository for register values. It comprises two register banks to maintain past register state (of the execution of predecessor tasks) and present register state (of the execution of the current task). The register control is responsible for coordination of communication and synchronization using information provided statically by the compiler and generated dynamically during execution. It performs this function via simple logical operations on a collection of bit masks: create mask, accum mask, recv mask, sent mask, recover mask, and squash mask. A brief definition of these mask is given, and the use of these masks is explained throughout the text that follows.

The *create mask* identifies all registers for which values may be created by the execution of a task. The *accum mask* is the combination of create masks for those tasks that logically precede a task. The *recv mask* identifies all register received during the execution of a task. The *kill mask* identifies all registers whose values are dead beyond the execution of a task. The *sent mask* identifies all register sent during the execution of a task. The *recover mask* identifies all registers for which correct values (to replace incorrect values created before a squash) must be provided by a task. The *squash mask* is the combination of recover masks for squashed tasks involved in the recovery process that precede a task.

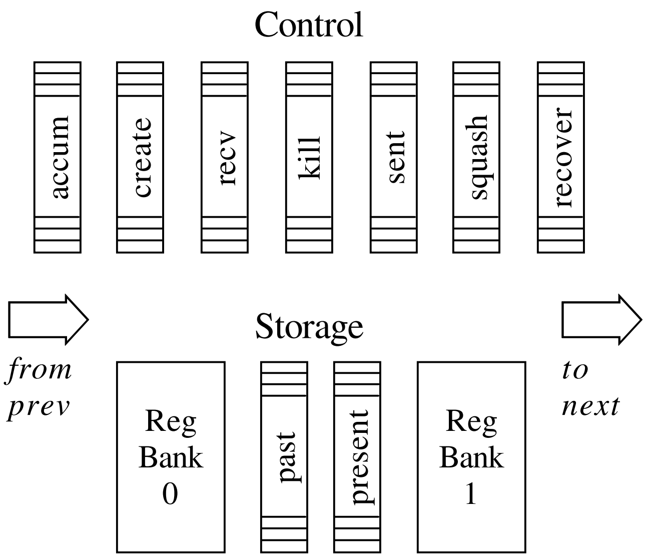


Figure 6.3: Basic register file components.

6.5.2.1 Storage

A register may be updated with a value from a predecessor task or a value from the current task. Conceptually, only a single set of register storage is required per task per processing unit, as the value produced by the current task logically displaces the value produced by the predecessor task. Yet, due to speculation it must be possible to recover at any time correct values for the task(s) on a processing unit whose register file(s) have been updated with incorrect values. The recovery may be performed by having the unaffected register files propagate the correct register values to the squashed register files. If it is not possible to rely on squashed register files to supply the precise identity of the registers to be recovered, the unaffected register files must by default send all registers. However, the register communication bandwidth required in this case is prohibitive. Indeed, it may be possible to rely on the squashed register files to identify the registers to be recovered. However, bidirectional communication between register files is required instead of unidirectional communication, which complicates the overall control structure.

A solution to the performance and complexity problems caused by one register set is to have two register sets, a *past* register set and a *present* register set. The past register set contains register values that have been created by predecessor tasks, and the present register set provides working storage for the register values of the current task. A *future* register set is not needed since values do not propagate beyond the task at the tail, since none of the compiler supplied information to coordinate register communication is available for tasks not yet seen. Logically, each register set has its own distinct storage; one bank of registers is the storage of the past set, and one bank of registers is the storage of the present set. Physically, if the past and present sets are separated into disjoint register banks, an expensive copy operation (of present to past) is required each time a task is committed. To avoid this multiple register copy operation, the register file may maintain a collection of pointers using additional control bits to provide the illusion of two distinct past and present register banks, somewhat similar to approaches proposed to implement boosting [84] and precise interrupts [69].

The control bits consist of two bit masks, the *past mask* and the *present mask*. The physical registers are divided into two banks, bank 0 and bank 1. The bit corresponding to register r_i in the past or present mask determines the physical bank which contains the corresponding instance of register r_i. At the time a task is assigned to processing unit, the past and present masks of the register storage for that task are identical. As the current task executes, the past and present masks are inspected and manipulated to coordinate register accesses. A register read inspects the present mask to access the appropriate register bank. A register write inspects the past mask (and chooses the other bank) to access the appropriate register bank and updates the present mask to reflect the bank which contains the present value of the register. In this way, register values from the past and present are kept distinct. At the time a task is committed, the past mask is updated with the present mask. In the event a task is squashed, the present mask is updated with the past mask, effectively discarding the (incorrect) present registers and retaining the (correct) past registers.

6.5.2.2 Communication

As a task executes, it produces register values. The *create mask* of a task identifies all registers for which values may be created by the execution of a task. The *kill mask* of a task identifies all registers whose values are dead beyond the execution of a task. Those registers on the create mask but not on the kill mask need to be forwarded to succeeding tasks (and processing units). In particular, the last instruction that writes into a register which is live outside the current task needs to propagate the register value. (An earlier instruction, for example any instruction that writes a register specified in the create mask, could also forward a register value, but this increases the chances of a later task being squashed from the use of an incorrect register value.) The indication of which instruction is the last one to update a register in a task is provided by the compiler (as already described). Nonetheless, the last instruction in a task to update a register can be determined in a straightforward manner from the control flow graph. (If no indication is available regarding when a register value ought to be propagated, the task might have to wait until the end of its execution to propagate the values of registers specified in the create mask.) Each register that propagates on the ring comprises a value, a register identifier, and a tag (of $\log_2(n)$ bits, where n is the maximum number of active tasks) that identifies the active task that produced it.

As the register propagates, it is written into the past register set for the current register file and is propagated to the next register file as determined by the register communication information for the task in execution. The passage of register values between register files is recorded on the *recv mask* and the *sent mask* of a task. The appropriate bit in the recv mask is set to indicate that the register value has been received by the current register file; the appropriate bit in the sent mask is set to indicate that the register value has been sent to the next register file. A register value need not be propagated by a register file if the task in execution creates another instance of the same register. As per sequential execution semantics, the more recent value is the one that is to be used by successor tasks. If the passage of a register value is not blocked by a register file, it propagates around the ring until it reaches the register file where it was originally created. Nevertheless, a register value can only propagate among the active tasks on the processor, which implies that register values must wait at the tail (until such time as the tail advances). Accordingly, each register propagates at most as many times as the maximum number of active tasks, although most register values propagate far less, depending upon the characteristics of the tasks [28].

6.5.2.3 Speculation

In the event of incorrect execution due to speculation, a mechanism must be in place to recover the proper register state for each register file. In the most basic terms, any register which has been created by a task that is a part of the incorrect execution must be recovered. The recovery actions are simple for incorrect register values that have been contained within a register file; the two bank register storage allows all incorrect registers to be discarded from the present storage with the past storage preserving the correct values. As might be expected, the recovery actions are more complex for incorrect register values that have been propagated from a register file (to other register files). In any event, the goal is to recover the minimal amount of register state to afford correct execution.

The difficulty that arises in recovery is not identifying which registers must be recovered; rather, it is discovering precisely which other register files units have been modified by incorrect register values. Without analyzing the aggregate register state information from all register files at the same time, optimal recovery is not possible. In general, though, correct recovery does not require this level of detail. Instead, recovery can be performed, not all at once, but distributed over time, as register files corresponding to squashed tasks are assigned tasks again. To approach the problem, it is necessary to identify the extent of the incorrect execution. The extent may be broken down into what register files have been affected and into what registers within the register files have been affected.

The affected register files are those register files for the task from the point of the squash to the task at the tail. The task at the point of the squash and the task at the tail are recorded in the prediction hardware as the “squash” head and “squash” tail respectively. The actual head remains as it is, and the actual tail is changed to the task at the point of the squash (as this point is where task assignment resumes). Thus, the register files corresponding to the tasks between the squash head and the squash tail are to be recovered. No other register files participate in recovery because no other register files could have been affected by the incorrect execution, since registers do not propagate beyond the task at the tail (for the reasons described above).

The determination of which registers to recover can be made via simple logical operations on the register control bit masks. For each squashed task, all registers which have been produced and propagated from it are placed on a *recover mask*. That is, the recover mask indicates the registers for which the register file must propagate correct values to replace the incorrect ones. The register file assumes the role of creator for each register to ensure that all register files see the corrective update. A *squash mask* is provided to synchronize the use of recovered values. The squash mask is the combined recover masks of all predecessor register files involved in the recovery process.

The squash mask is transferred along to each register file as tasks are assigned to replace the tasks that were squashed. As each task is assigned to replace the one that was squashed, the corresponding register file receives a squash mask from its immediate predecessor register file and removes any registers in the squash mask from its recv mask. To assemble the squash mask (for the register file for the next task to be assigned), the register file produces a recover mask (of its prior incorrect execution) and performs the bitwise OR of the squash mask and this recover mask. The registers in the squash mask of the sum are removed from the sent mask of the register file. There is not difficulty with respect to a means to set bits, but no means to clear bits (as well be described later for the accum mask), because its scope is confined to the register files between the squash head and squash tail.

In the event another squash occurs before an outstanding squash has been entirely recovered, it is not the case that the recovery information for the new squash is stacked upon the recovery information for prior (possibly multiple) outstanding squashes. Such an approach may require the assemblage of an unbounded quantity of state. Instead, the recovery information for the new squash is integrated directly into the recovery information for the prior outstanding squashes. Any registers which have been recovered need not be recovered again (unless required by the new squash). Any registers which have not been recovered are added to the existing recover mask.

6.5.2.4 Synchronization

To provide synchronization between tasks, the control in each register file must identify if and when the correct register values for an instruction to execute are available. In particular, the control must (i) recognize what register values are produced by all predecessor tasks, (ii) determine if these register values are still in-flight, and (iii) wait until such time as these register values are received. Though it might seem that these are complicated functions in terms of control, it is possible to provide them in a straightforward manner using information provided statically by the compiler and gathered dynamically during execution. In particular, the register control may reduce these seemingly complex functions into simple logical operations performed on a collection of static and dynamic bit masks (as described below).

The create mask of a task is not sufficient to coordinate the production and consumption of register values among all (possibly non-adjacent) tasks. As a task may require register values from any one of its predecessors, only the combined create masks of all predecessors supplies the required information. This combination of create masks is assembled into the *accum mask* (or cumulative create mask) of a task. The accum mask is provided to a task being assigned by its immediate predecessor. To assemble the accum mask (for the next task assigned), a register file performs the bitwise OR of the accum mask and the create mask of the current task. The accum mask synchronizes the consumption of registers in the current task with the production of registers in the predecessor tasks. Of the registers indicated on the accum mask, any that have not yet been received (indicated by the recv mask) are busy. A read of a busy register may require a stall in the processing unit; a write of a busy register never stalls, as it produces, rather than consumes a register value.

An important consideration with regard to the mechanism defined above is that it provides a means to set bits in the accum mask, but no means to clear bits in the accum mask. In short order, all bits of the accum mask would become set. As such, any task assigned would expect to wait in all cases for all registers. To deliver the desired semantics of this strategy, any bit of the accum mask that corresponds to a register instance that has made a complete cycle back to its creator must be cleared at the next task assignment. In order to provide this functionality, each bit of the accum mask must be tagged with its corresponding creator processing unit. The only drawback of this technique is that $\log_2(n)$, where n is the maximum number of active tasks, times as many bits are required to represent the accum mask, and hence must be communicated between register files at task assignment. An alternative to this technique (not considered further here) is to produce the accum mask by combining create masks unchecked in the hardware, and rely only on the compiler to remove dead registers (via the kill mask) for each task.

6.6 Register File Evaluation

The register file evaluation is divided into two parts, one a behavior part and the other a performance part. The behavior part is intended to give a feel for the demands the register aspect of the data supply places upon the register file during execution. The performance is intended to show how the register file design studied here affects performance with different bandwidth and latency characteristics.

First, the behavior part is presented. It consists of some basic measurements of register communication behavior during execution. In particular, the register traffic per useful task as well as the register traffic per useful instruction are shown to indicate how much register communication occurs between the register files of adjacent processing units. In addition, the total register traffic per cycle and link register traffic per cycle as well as the register travel distance in terms of the number of tasks are given to identify the potential impact of bandwidth and latency on register communication.

Second, the performance part is presented. It consists of an evaluation of the performance impact of varying bandwidth and latency for the register communication links between adjacent register files on the point-to-point unidirectional communication ring. Though there are both intra-task and inter-task aspects to communication via the register file, this performance evaluation only studies the impact of bandwidth and latency on inter-task register communication. The bandwidth and latency of intra-task register communication is handled with no undue impact on performance in the same manner as for conventional processors.

6.6.1 Metrics

There are a number of metrics used here to consider the behavior of register communication during execution. Even though the metrics are different from one another, this evaluation provides all of these register traffic measures as unweighted arithmetic means (AMEAN) for each of the two groups of benchmarks, SPEC CINT95 and SPEC CFP95, respectively. For all of these metrics, the arithmetic (as opposed to the harmonic or geometric) mean is the correct mean because these measures do not necessarily have a direct correlation to program execution time, but instead are simply counts of register behavior; the mean should not be weighted, as in all other cases in this thesis, since the type of (integer or floating point) application rather than the number of instructions each benchmark program executes is the relevant factor. The individual metrics are described in text that follows.

First, the register traffic per useful task and the register traffic per useful instruction is given. The register traffic is the observed register communication from each register file to the next adjacent register file (on the links of the communication ring) summed among all register files. It includes register instances generated by a task and all previous tasks that must be communicated. This traffic is then divided by the number of useful tasks or useful instructions. Second, the total register traffic per cycle and the link register traffic per cycle is given. The total register traffic per cycle is the observed register traffic over all communication links divided by the number of program execution cycles. The link register traffic per cycle is the observed register traffic per communication link divided by the number of program execution cycles. Third, the register task travel distance is given. The register task travel distance is the distance that a register instance travels from the task that generates it, counted in terms of tasks from the dynamic instruction sequence.

Though the behavior of register communication during execution is useful to consider the demands placed on the register file, it does not by itself indicate what effect the register file may have in terms of overall performance, which (as has been asserted for the other processor components) is the true test of any processor component design. The actual processor performance for register file that has different bandwidth and latency characteristics can give an indication of the extent to which the ability of the processor to extract instruction-level parallelism is determined by the ability of the register file to supply register data. As has been the case throughout this thesis, the performance given as the unweighted harmonic mean (HMEAN) of useful instructions per cycle (discussed in Section 4.6.1) for each of the two groups of benchmarks, the SPEC CINT95 and the SPEC CFP95, is used for this purpose.

6.6.2 Configurations

Most of the other processor components are fixed to the standard configuration presented in Table 3.11 of Chapter 3 for this evaluation.

Yet, for all of the register file configurations studied, a range of processing unit configurations are used. In particular, the number of processing units is varied among 2, 4, 8, and 16, as well as among 4 processing units running 2 and 4 tasks one-at-a-time and all-at-once. With respect to the register file, its configuration is fixed for the behavior part of the evaluation and varied for the performance part.

For the behavior part, the register file is configured with a bandwidth of 4 registers communicated per cycle between adjacent processing units and a latency of 1 cycle for the communication between processing units. Thus, as many as 4 register values may be produced by a processing unit each cycle and consumed by an adjacent processing unit the next cycle, with an additional cycle of latency for each processing unit thereafter.

For the performance part, the register file is configured with bandwidth varying among 1, 2, 3, and 4 register values communicated per cycle; the register file is configured with latency varying among 1, 2, 3, and 4 cycles for the communication of register values between adjacent processing units. Moreover, it is assumed that the register communication is pipelined between register files, so that register communication may be initiated each cycle.

6.6.3 Behavior

The purpose of studying the behavior of the register file is to provide a feel for the nature of register communication in a decentralized register file design and to give an indication of the effect on overall performance that might accompany changes in the bandwidth and/or latency characteristics of the register file. The behavior results are separated according to the particular aspect of register traffic under consideration. As described above, the register traffic metrics may be divided among three aspects. To begin, the register traffic per useful task and the register traffic per useful instruction are presented. Then, the total register traffic and the link register traffic are shown. To end, the register task travel distance is considered.

6.6.3.1 Register Traffic Per Useful Task and Instruction

Each of the graphs in the figures referred to below, Figures 6.4, 6.5, 6.6, and 6.7, plots the register traffic with and without mis-speculation. The plots with mis-speculation reflect the behavior of register communication that takes place under realistic execution conditions. The plots without mis-speculation reflect the behavior of register communication that occurs under idealistic execution conditions, since these are generated by using perfect information that prevents control and data dependence violations from occurring.

In addition, each graph plots (as a dashed line) the number of register instances generated per useful task or per useful instruction (as the case may be) as a point of reference. Since the register traffic accounts for register instances from all previous tasks as well as those from a task itself, this line represents the minimum amount of register traffic, in which all registers propagate no farther than the next task.

Figure 6.4 gives the register traffic per useful task (on the left) and per useful instruction (on the right) for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Figure 6.5 gives the same SPEC CINT95 results running 2 and 4 tasks per processing unit, one-at-a-time and all-at-once, using the 4 processing unit configuration.

Consider the register traffic per useful task and the register traffic per useful instruction for the SPEC CINT95 benchmarks. For both metrics, per useful task and per useful instruction, the register traffic increases steadily as the number of tasks in concurrent execution is increased. Moreover, the amount of register traffic per useful task and per useful instruction exceeds the traffic due solely to the register instances generated, from as much as 2.0 registers per useful task and 0.2 registers per useful instruction for 2 processing units to as much as 10.0 registers per useful task and 0.9 registers per useful instruction. Despite the fact that the register traffic does increase as the number of processing units increases, it is significant that the magnitude of these increases is not proportional.

Even though the number of tasks in concurrent execution is doubled for the single task per processing unit configurations in Figure 6.4, the register traffic does not increase at nearly this rate. Likewise, for the multiple tasks per processing unit configurations in Figure 6.5, the register traffic does not increase at the same rate as the number of tasks is doubled, nor does it increase a significant amount from running one-at-a-time to all-at-once. The reason for this behavior is that many of the register instances generated by earlier tasks are filtered by later tasks as other register instances of the same register name are generated. Moreover, it must be noted that the plots with mis-speculation and without mis-speculation diverge as the number of processing units increases because mis-speculation occurs more often with more processing units, so there is more traffic generated as a result, a difference of nearly 3.0 registers per useful task and 0.3 registers per useful instruction for 16 processing units.

Figure 6.6 gives the register traffic per useful task (on the left) and per useful instruction (on the right) for the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Figure 6.7 gives the same SPEC CFP95 results running 2 and 4 tasks per processing unit, one-at-a-time and all-at-once, using the 4 processing unit configuration.

Consider the register traffic per useful task and the register traffic per useful instruction for the SPEC CFP95 benchmarks. The overall behavior for the floating point benchmarks is

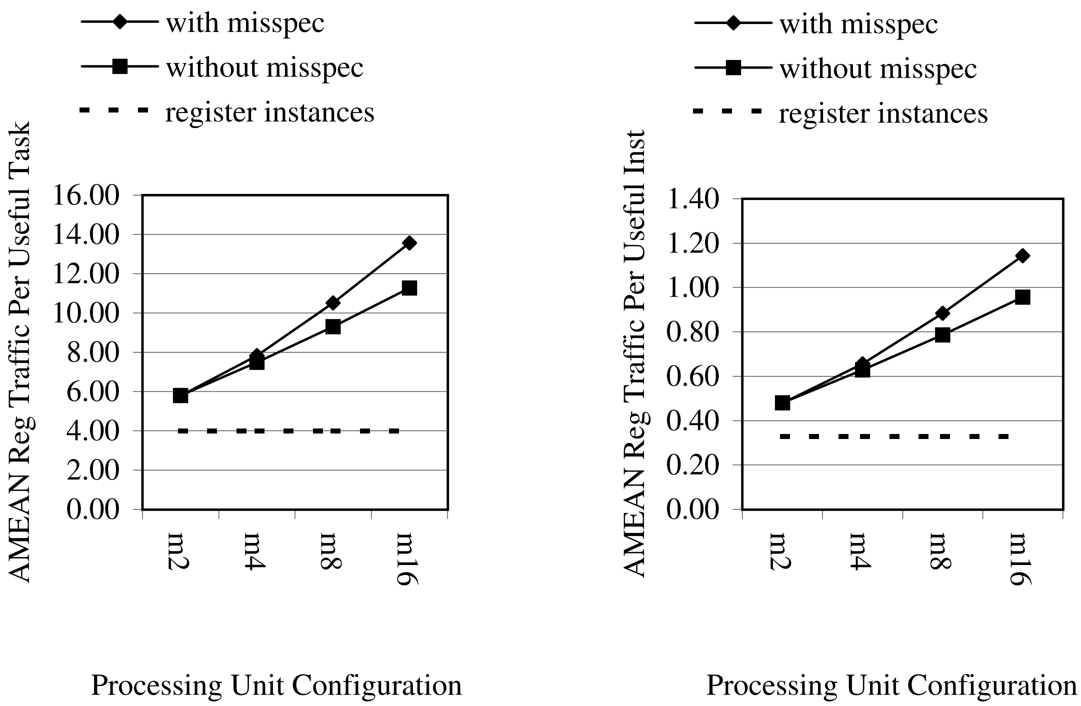


Figure 6.4: SPEC CINT95 register traffic per useful task and register traffic per useful instruction for 2, 4, 8, and 16 processing units.

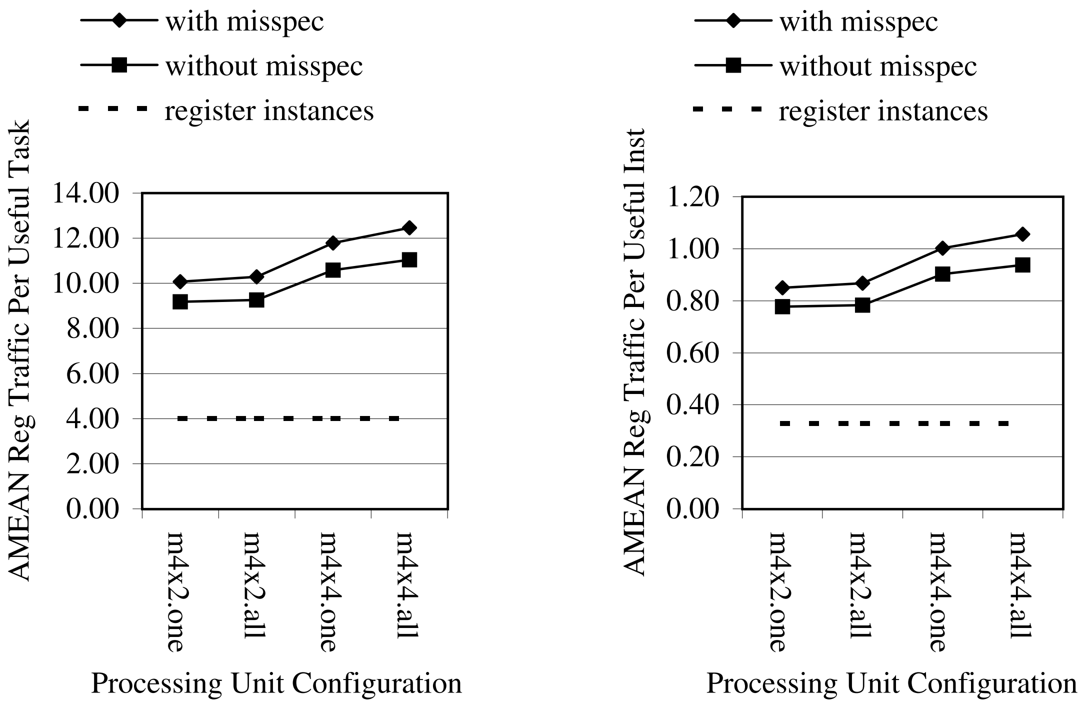


Figure 6.5: SPEC CINT95 register traffic per useful task and register traffic per useful instruction for 4 processing units running 2 and 4 tasks, one-at-a-time and all-at-once.

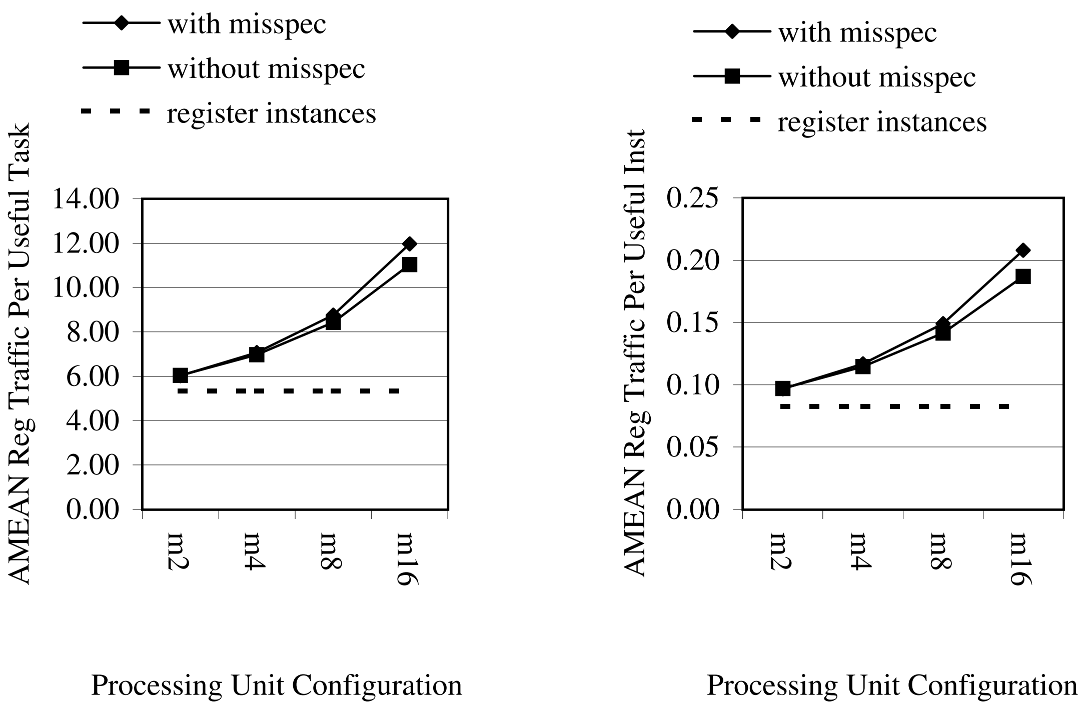


Figure 6.6: SPEC CFP95 register traffic per useful task and register traffic per useful instruction for 2, 4, 8, and 16 processing units.

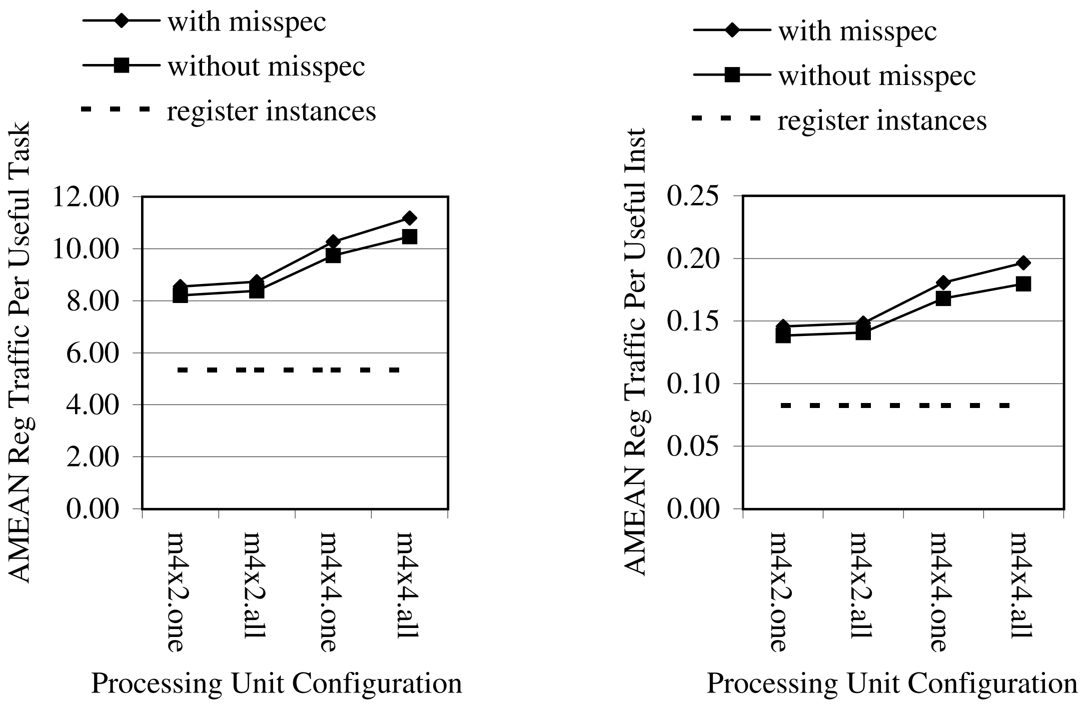


Figure 6.7: SPEC CFP95 register traffic per useful task and register traffic per useful instruction for 4 processing units running 2 and 4 tasks, one-at-a-time and all-at-once.

similar to that of the integer benchmarks for configurations running a single task per processing unit as shown in Figure 6.6 and for configurations running multiple tasks per processing unit as shown in Figure 6.7. The register traffic increases steadily as the number of tasks in concurrent execution is increased for both the register traffic per useful task and per useful instruction. Likewise, the amount of register traffic exceeds what is due to the register instances generated. Nevertheless, the increases in absolute terms are comparable for the floating point and integer benchmarks for register traffic per useful task. Yet, the increases in absolute terms for register traffic per useful instruction are much less for floating point than for integer benchmarks (around 0.08 registers per useful instruction going from 2 to 16 processing units running a single task for floating point and at least 0.50 for integer, with similar behavior for processing unit configurations running multiple tasks).

There are some other differences between the integer and floating point benchmarks. First, the divergence of the amount of register traffic with mis-speculations and without mis-speculations as the number of processing units is increased is much narrower for the floating point benchmarks than for the integer benchmarks, *e.g.* a difference of only 1.0 registers per useful task as compared to 3.0 registers per useful task. For the most part, this difference in behavior is due to the fact that fewer control and/or data dependence violations occur in the floating point programs. As a result, the realistic behavior is much closer to the idealistic behavior for the floating point than for integer programs. Second, the register traffic for floating point programs is much closer to the number of register instances than for integer programs. Again, part of this behavior is due to fewer dependence violations. However, part of it is also due to the fact that registers do not travel as far for the floating point benchmarks as for the integer ones (as discussed next).

6.6.3.2 Total and Link Register Traffic

Each of the graphs in the figures referred to below, Figures 6.8, 6.9, 6.10, and 6.11, plots the register traffic with and without mis-speculation, as in the previous graphs. The plots with mis-speculation reflect the behavior of register communication that occurs under realistic execution conditions. The plots without mis-speculation reflect the behavior of register communication that takes place under idealistic execution conditions, since these are generated by using perfect information that prevents control and data dependence violations from occurring. It is worth reiterating that the total register traffic per cycle is for the entire processor, over all communication links; the link register traffic per cycle is just for each link between adjacent processing units.

Figure 6.8 gives the total register traffic per cycle (on the left) and link register traffic per cycle (on the right) for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Figure 6.9 gives the same SPEC CINT95 results running 2 and 4 tasks per processing unit, one-at-a-time and all-at-once, using the 4 processing unit configuration.

Consider the total register traffic per cycle and the link register traffic per cycle for the SPEC CINT95 benchmarks. In the case of the total register traffic per cycle it may be seen that the traffic per cycle increases steadily as the number of tasks in concurrent execution increases with and without mis-speculation. For configurations that use only a few processing units, the total register traffic per cycle with and without mis-speculation is about the same

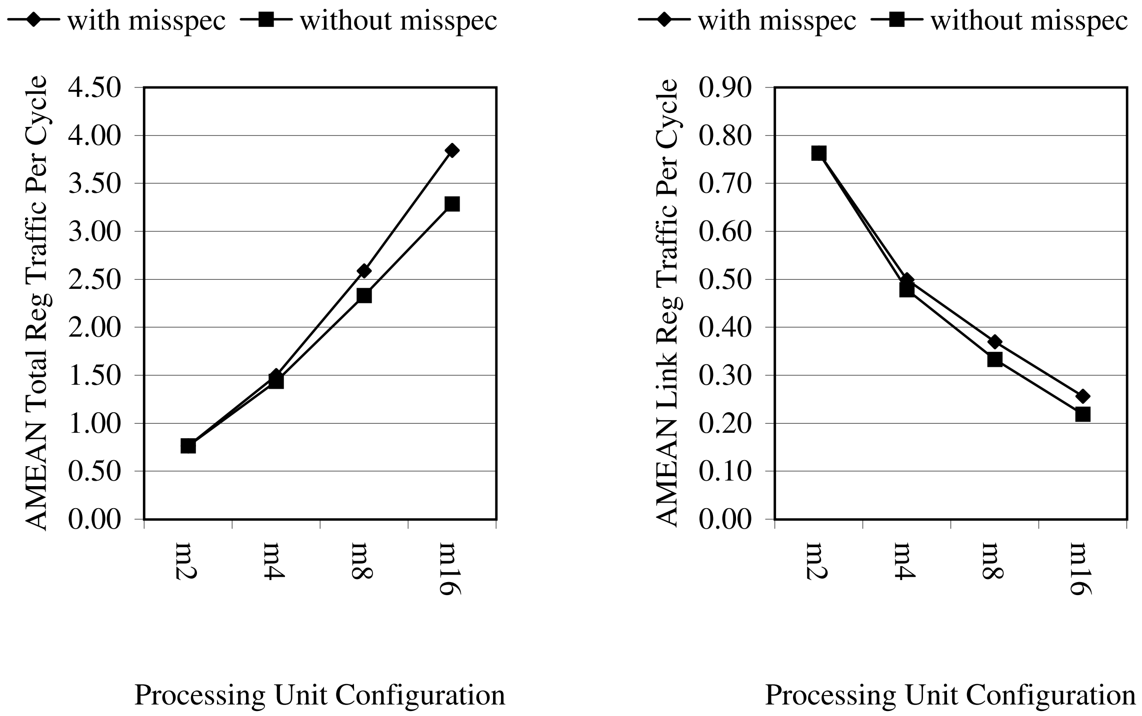


Figure 6.8: SPEC CINT95 total register traffic per cycle and link register traffic per cycle for 2, 4, 8, and 16 processing units.

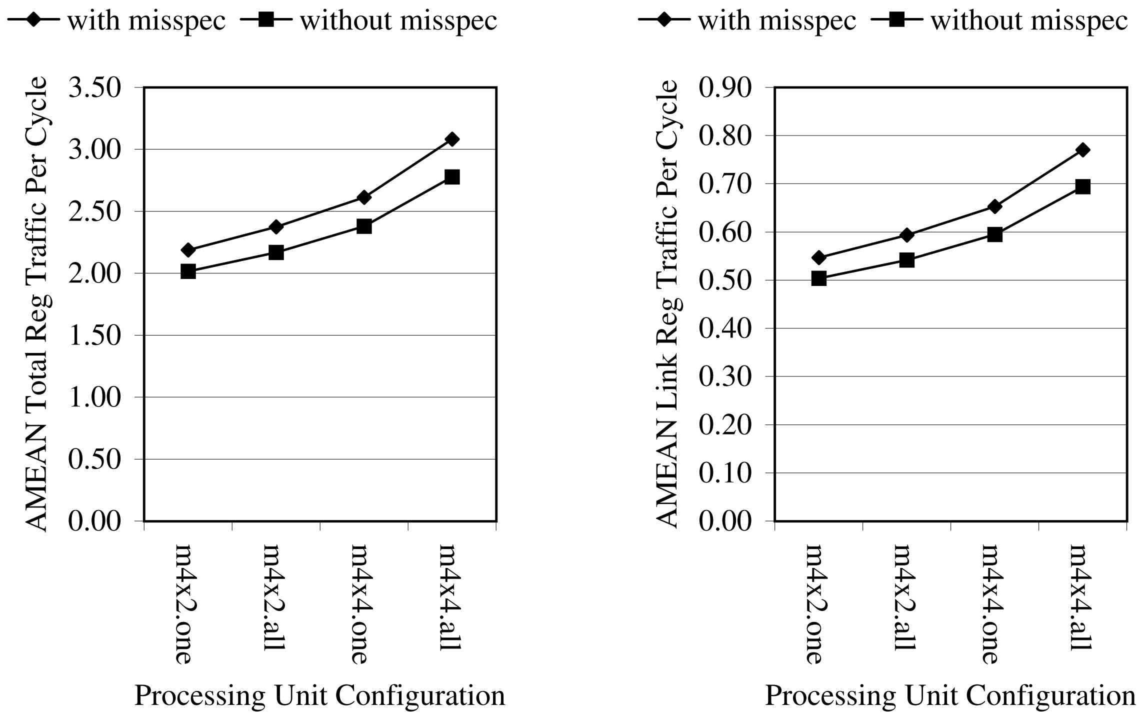


Figure 6.9: SPEC CINT95 total register traffic per cycle and link register traffic per cycle for 4 processing units running 2 and 4 tasks, one-at-a-time and all-at-once.

since control and data dependence violations are less likely for fewer processing units. For configurations with many tasks, either one per processing unit or many per processing unit, the total register traffic per cycle is greater with than without mis-speculation, but not even by as much as 1.0 registers per cycle in the worst case.

However, for single task per processing unit configurations the link register traffic per cycle actually decreases as the number of tasks increases. The reason for this behavior is that the number of processing units, and more significantly the number of communication links, increases as does the number of tasks. Yet, because the total register traffic does not increase at the same rate as the number links, the link register traffic decreases. In contrast, for multiple tasks per processing unit configurations the link register traffic per cycle indeed increases as the number of tasks increases, but only slightly. The reason for this behavior is that the number of processing units, and likewise the number of communication links, does not change as does the number of tasks. So, because the total register traffic increases, the link register traffic increases as well.

Figure 6.10 gives the total register traffic per cycle (on the left) and link register traffic per cycle (on the right) for the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Figure 6.11 gives the same SPEC CFP95 results running 2 and 4 tasks per processing unit, one-at-a-time and all-at-once, using the 4 processing unit configuration.

Consider the total register traffic per cycle and the register task travel distance for the SPEC CFP95 benchmarks. The overall behavior trends for both metrics are more or less the same for the floating point benchmarks as for the integer benchmarks. Nonetheless, in absolute terms the differences are far more striking, particularly for configurations of many processing units. Looking at the case of total register traffic per cycle with mis-speculations when running a single task per processing unit, the 2 processing unit configuration for floating point benchmarks uses about 0.20 registers per cycle, while for integer benchmarks it uses more than 0.60 registers per cycle (more than 3 times as much); the 16 processing unit configuration use less than 1.40 registers per cycle for floating point and more than 3.60 registers per cycle for integer (more than $2\frac{1}{2}$ times as much). Looking at the case of total register traffic per cycle with mis-speculations when running multiple tasks per processing unit, similar behavior is observed. Of note, there is a significant difference in register traffic when the number of tasks per processing units goes from 2 to 4. In particular, the difference with mis-speculation and without mis-speculation is much more apparent.

6.6.3.3 Register Task Travel Distance

The graphs in Figures 6.12 and 6.13 plot the register task travel distance with and without mis-speculation, as in the previous graphs. Figure 6.12 gives the register task travel distance for the SPEC CINT95 benchmarks running 1 task per processing unit using 2, 4, 8, and 16 processing unit configurations (on the left) and running 2 and 4 tasks per processing unit, one-at-a-time and all-at-once, using the 4 processing unit configuration (on the right). Figure 6.13 gives the register task travel distance for the SPEC CFP95 benchmarks with the same processing unit configurations.

Again, the register task travel distance is the distance in terms of number of tasks from the dynamic instruction sequence that a register value travels. The plots with mis-speculation

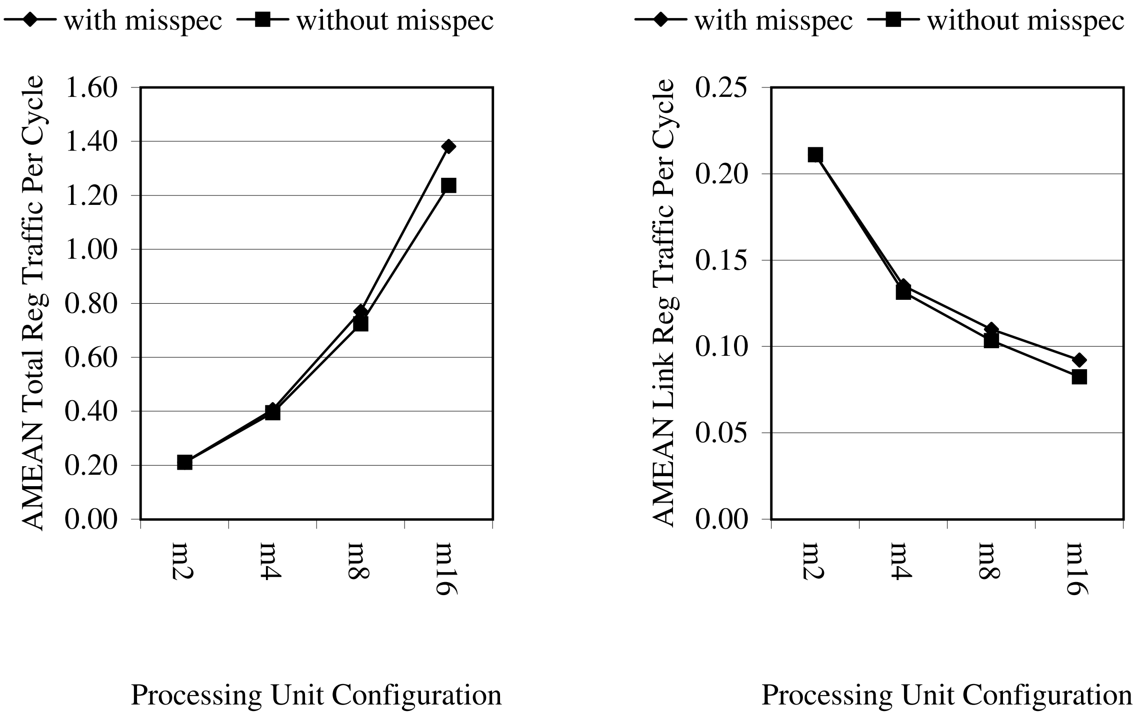


Figure 6.10: SPEC CFP95 total register traffic per cycle and link register traffic per cycle for 2, 4, 8, and 16 processing units.

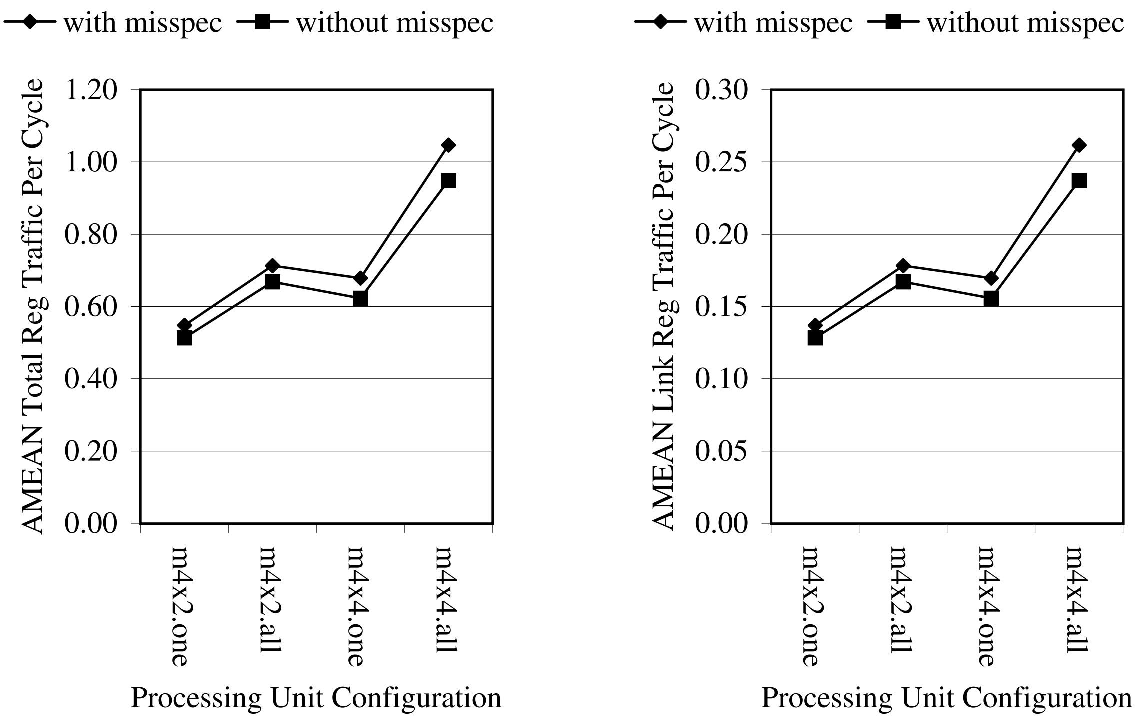


Figure 6.11: SPEC CFP95 total register traffic per cycle and link register traffic per cycle for 4 processing units running 2 and 4 tasks, one-at-a-time and all-at-once.

reflect the behavior of register communication that occurs under realistic execution conditions. The plots without mis-speculation reflect the behavior of register communication that takes place under idealistic execution conditions, since these are generated by using perfect information that prevents control and data dependence violations from occurring.

Comparing the integer and floating point benchmarks, registers travel farther as the number of tasks in concurrent execution increases, more so with mis-speculation than without, but registers still travel only a few “hops” (between 1 and 4) from the tasks that generate them. The distance in register link “hops” that a register travels may be as much as the number of tasks in concurrent execution (a full cycle of the tasks). Yet, as the figures show, the register task travel distance is held in check as the number of tasks in concurrent execution grows, even taking into account the effects of mis-speculation.

Contrasting the integer and floating point benchmarks, registers travel almost twice as far for the integer benchmarks as for the floating point benchmarks. This behavior is due in part to the fact that integer and floating point tasks are much different in sizes and in part due to the fact that the communication patterns of the integer and floating point benchmarks are intrinsically different. In addition, the difference between the distances with mis-speculation and without mis-speculation are somewhat more significant for integer than for floating point benchmarks.

6.6.4 Performance

The performance results are presented two ways, from a bandwidth perspective and from a latency perspective. In the bandwidth perspective presentation, the latency is fixed on each individual graph, while the bandwidth is varied to the show the effect it has on performance for a particular latency. In the latency perspective presentation, the bandwidth is fixed on each individual graph, while the latency is varied to show its effect on performance for a particular bandwidth. It is worth pointing out that this presentation actually gives the same results twice. However, the two perspectives make it easier to identify which register communication factor plays the more significant role in determining the impact of the register file on overall performance.

6.6.4.1 Bandwidth Perspective

Figure 6.14 gives the processor performance from a bandwidth perspective for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Figure 6.15 gives the same result for the SPEC CINT95 benchmarks running 2 and 4 tasks per processing unit, one-at-a-time and all-at-once, using the 4 processing unit configuration. Figure 6.16 gives the processor performance from a bandwidth perspective for the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Figure 6.17 gives the same result for the SPEC CFP95 benchmarks running 2 and 4 tasks per processing unit, one-at-a-time and all-at-once, using the 4 processing unit configuration.

Each of the figures contain four graphs. The register file latency, fixed for all results that appear on each graph, is indicated at the top of each graph. For the integer benchmarks, the same performance trends may be seen as the register file bandwidth is improved, for

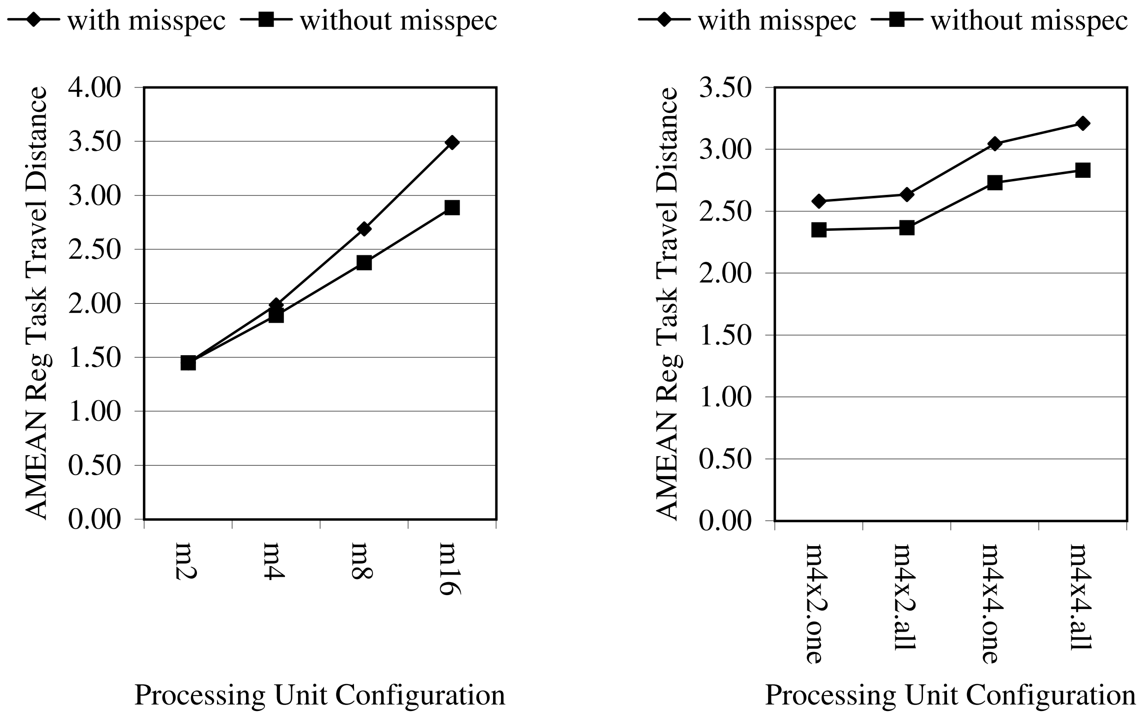


Figure 6.12: SPEC CINT95 register task travel distance for 2, 4, 8, and 16 processing units and for 4 processing units running 2 and 4 tasks, one-at-a-time and all-at-once.

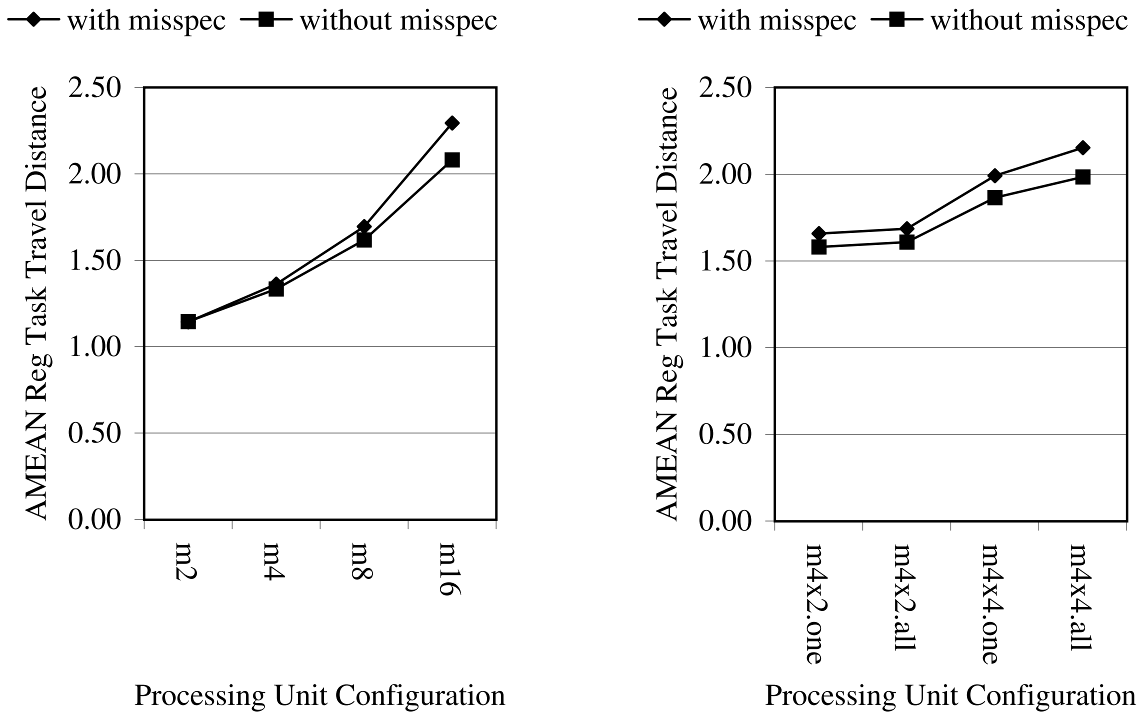


Figure 6.13: SPEC CFP95 register task travel distance for 2, 4, 8, and 16 processing units and for 4 processing units running 2 and 4 tasks, one-at-a-time and all-at-once.

each different register file latency. The overall performance readily increases in going from a register bandwidth of 1 register per cycle to 2 registers per cycle, but quickly levels off in going above 2 registers per cycle. For the floating point benchmarks, again the same performance trends may be seen as register bandwidth is improved, for each different register file latency. However, the overall performance has little or no relative increases in going above a register bandwidth of 1 register per cycle.

6.6.4.2 Latency Perspective

Figure 6.18 gives the processor performance from a latency perspective for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Figure 6.19 gives the same result for the SPEC CINT95 benchmarks running 2 and 4 tasks per processing unit, one-at-a-time and all-at-once, using the 4 processing unit configuration. Figure 6.20 gives the processor performance from a latency perspective for the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Figure 6.21 gives the same result for the SPEC CFP95 benchmarks running 2 and 4 tasks per processing unit, one-at-a-time and all-at-once, using the 4 processing unit configuration.

Each of the figures contain four graphs. The register file bandwidth, fixed for all results that appear on each graph, is indicated at the top of each graph. For the integer benchmarks, the overall performance decreases steadily and at significant rate as the register file latency is increased. Furthermore, this drop in performance is more precipitous the more aggressive the processor configuration, in terms of number of processing units. In contrast, for the floating point benchmarks, the overall performance still decreases steadily, but at relatively limited rate, except for the most aggressive processor configuration of 16 processing units.

6.6.4.3 Putting It All Together

Overall, it seems clear that the register file latency rather than the register file bandwidth is the critical factor in terms of the impact of the register file on processor performance. In particular, the register file latency has the effect of compressing the performance of the more aggressive processing unit configurations to that of the performance of the less aggressive processing unit configurations, more so for integer than for floating point benchmarks. There is a similar effect due to the register file bandwidth, but it is relatively insignificant in this study. Thus, the register file bandwidth does have an effect on processor performance, though it is much less significant relative to the register file latency. To be specific, the processor performance improves steadily and at a fairly high rate as the register file latency is decreased; yet, the processor performance only improves marginally as the register bandwidth is increased and at a very small rate that levels off rather quickly.

6.7 Data Memory Design

The data memory must provide high bandwidth to sustain multiple concurrent memory references per cycle and low latency to avoid aggravating dependences within and among tasks.

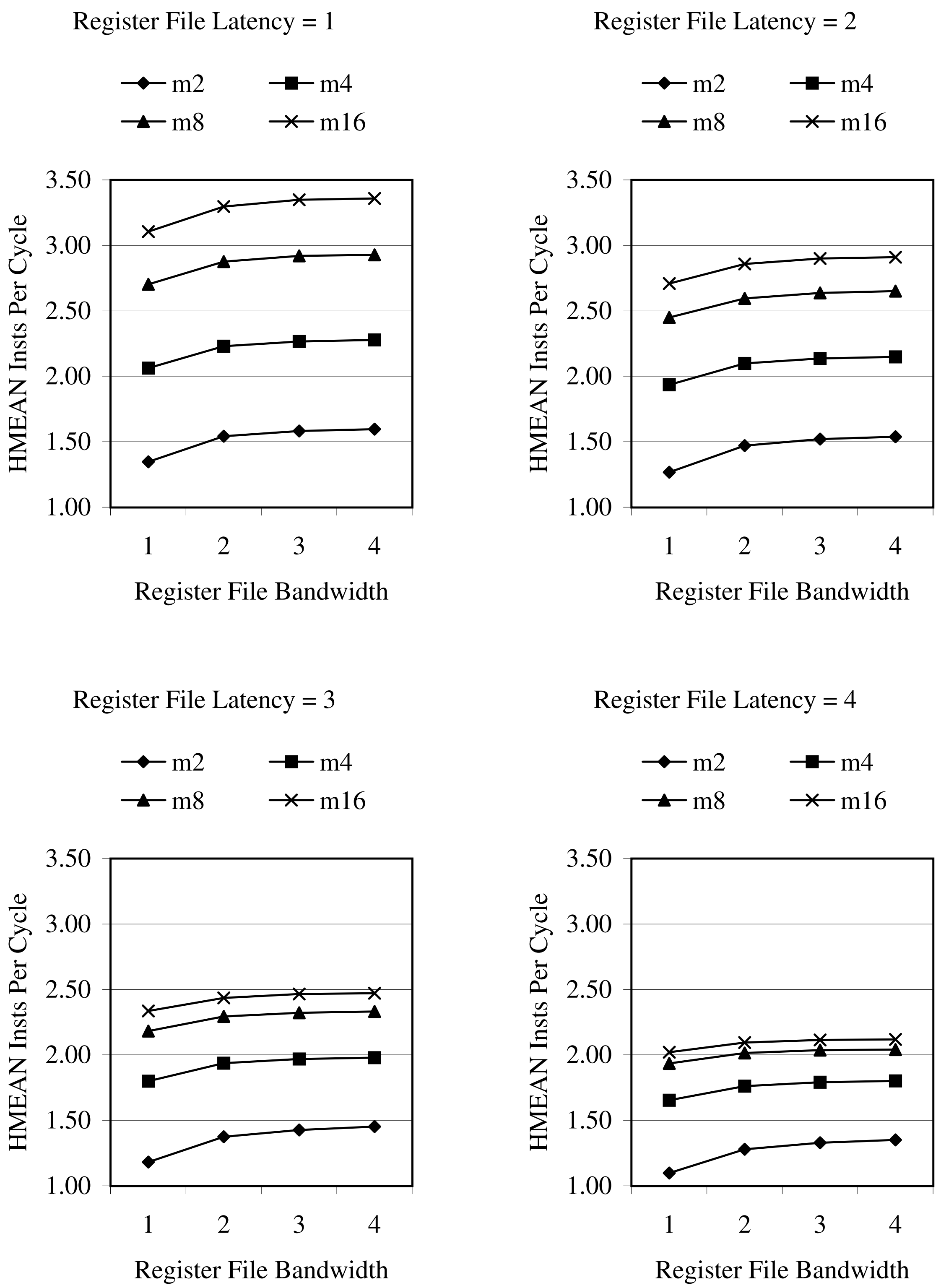


Figure 6.14: SPEC CINT95 performance of 1, 2, 3, and 4 cycle register latency with 1, 2, 3, and 4 register bandwidth for 2, 4, 8, and 16 processing units.

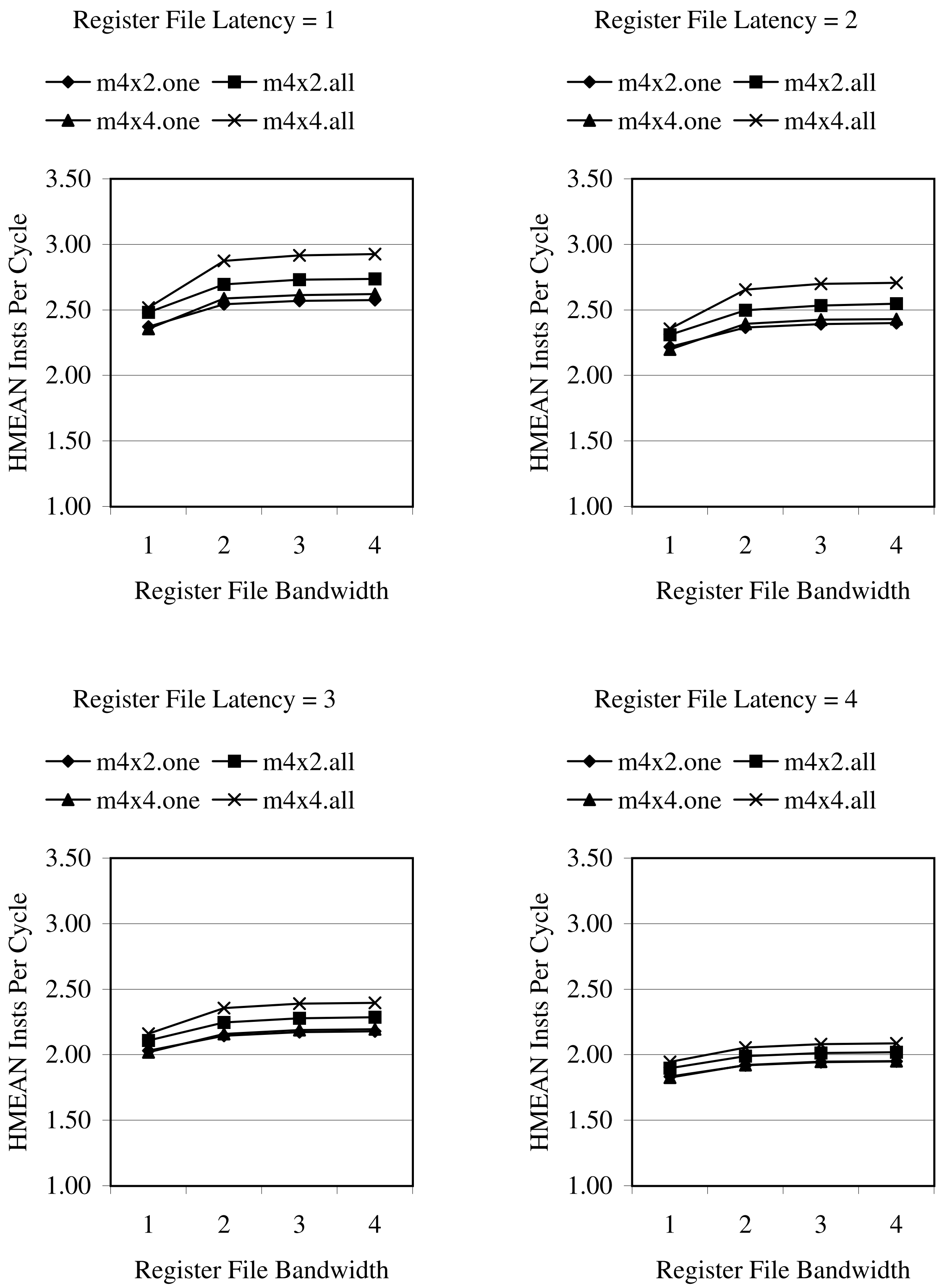


Figure 6.15: SPEC CINT95 performance of 1, 2, 3, and 4 cycle register latency with 1, 2, 3, and 4 register bandwidth for 4 processing units running 2 and 4 tasks, one-at-a-time and all-at-once.

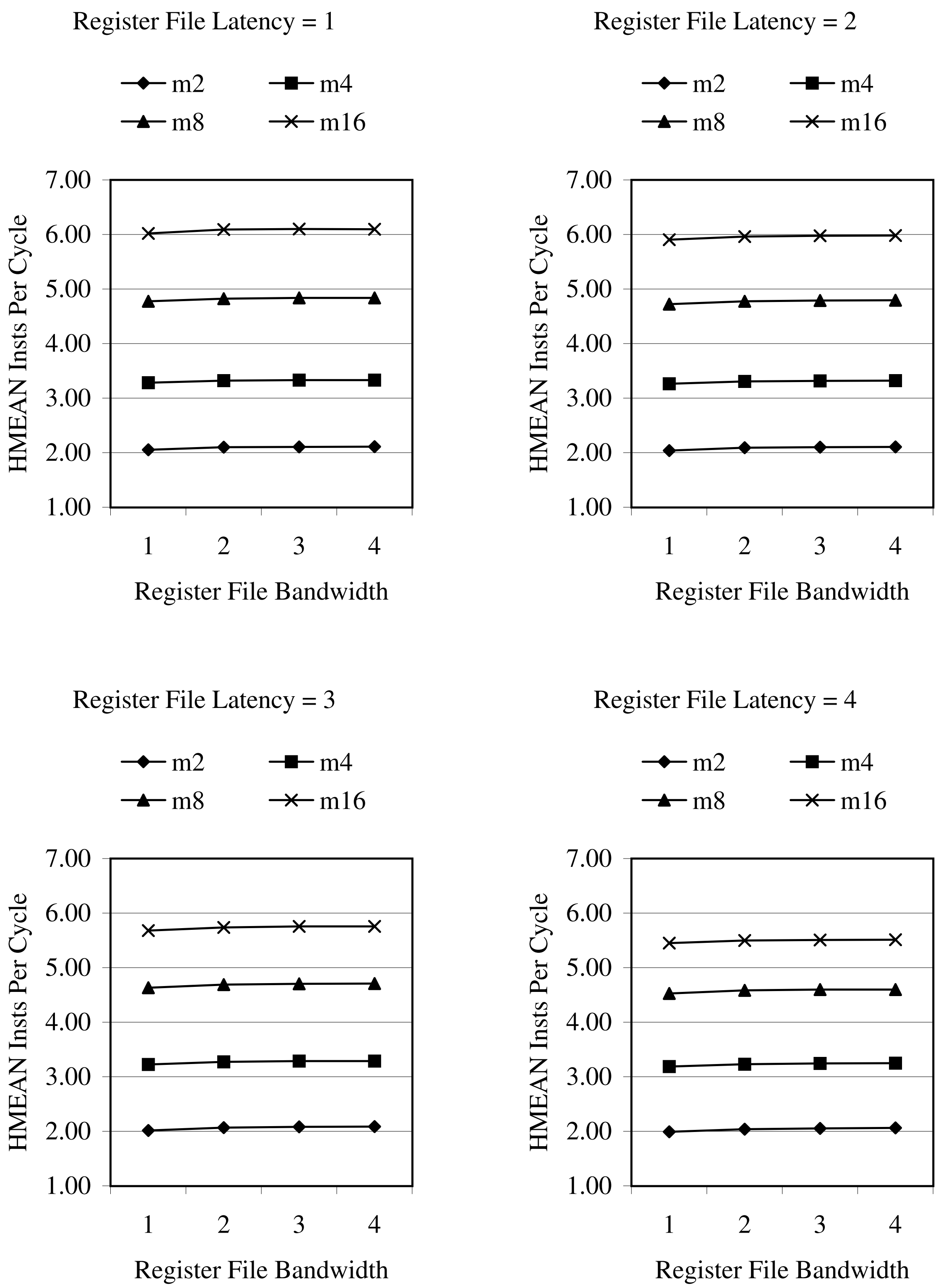


Figure 6.16: SPEC CFP95 performance of 1, 2, 3, and 4 cycle register latency with 1, 2, 3, and 4 register bandwidth for 2, 4, 8, and 16 processing units.

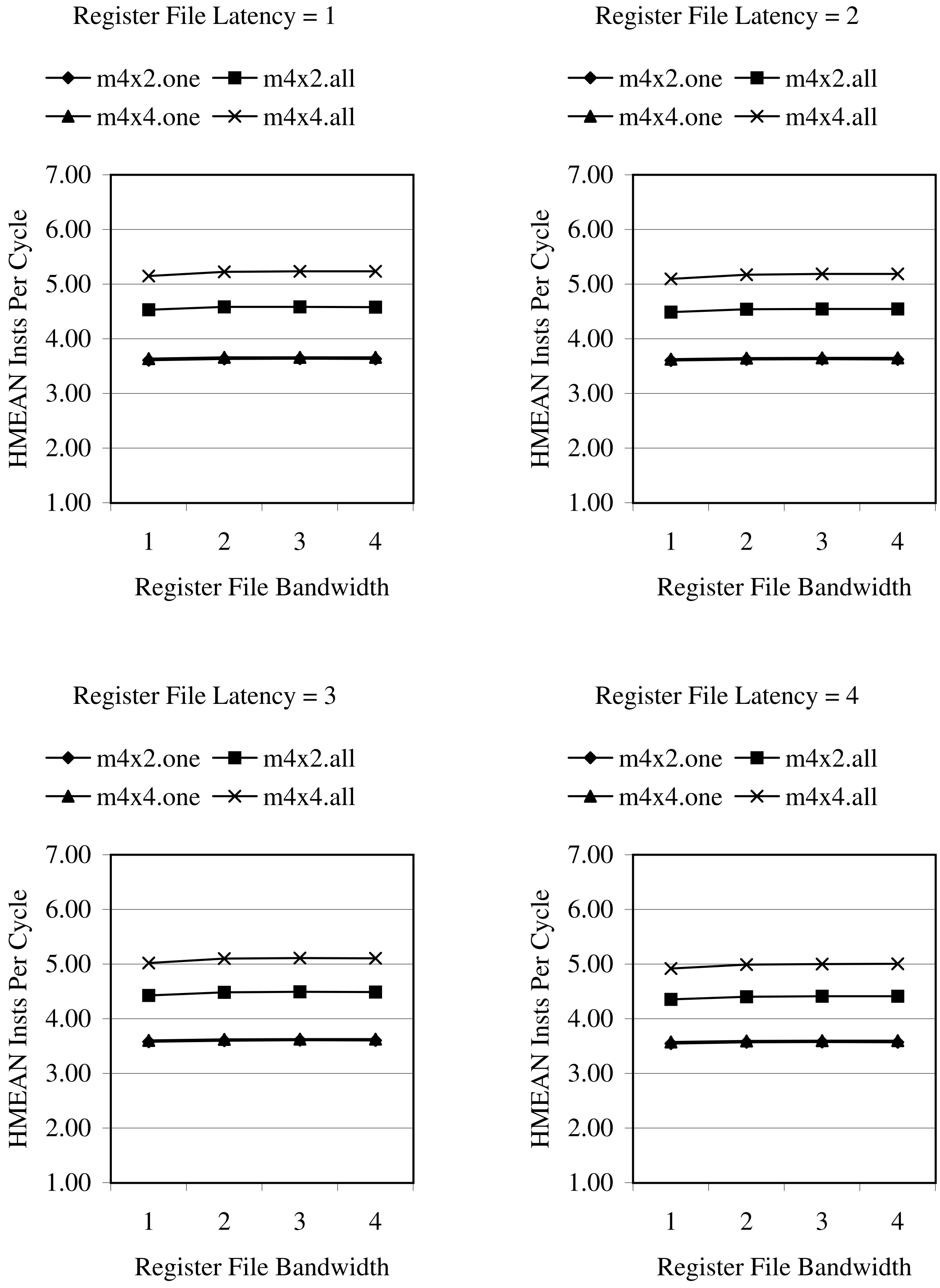


Figure 6.17: SPEC CFP95 performance of 1, 2, 3, and 4 cycle register latency with 1, 2, 3, and 4 register bandwidth for 4 processing units running 2 and 4 tasks, one-at-a-time and all-at-once.

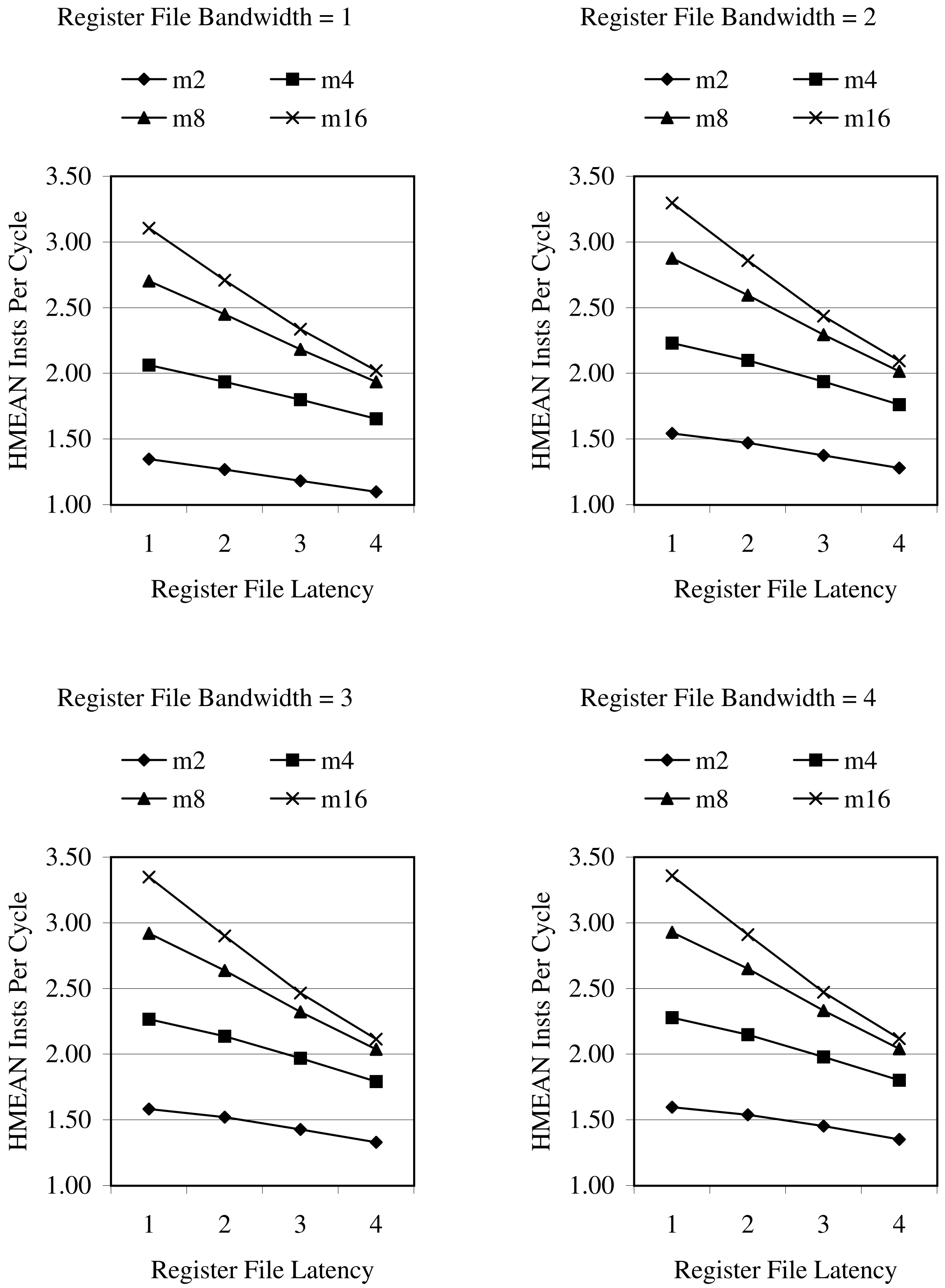


Figure 6.18: SPEC CINT95 performance of 1, 2, 3, and 4 register bandwidth with 1, 2, 3, and 4 cycle register latency for 2, 4, 8, and 16 processing units.

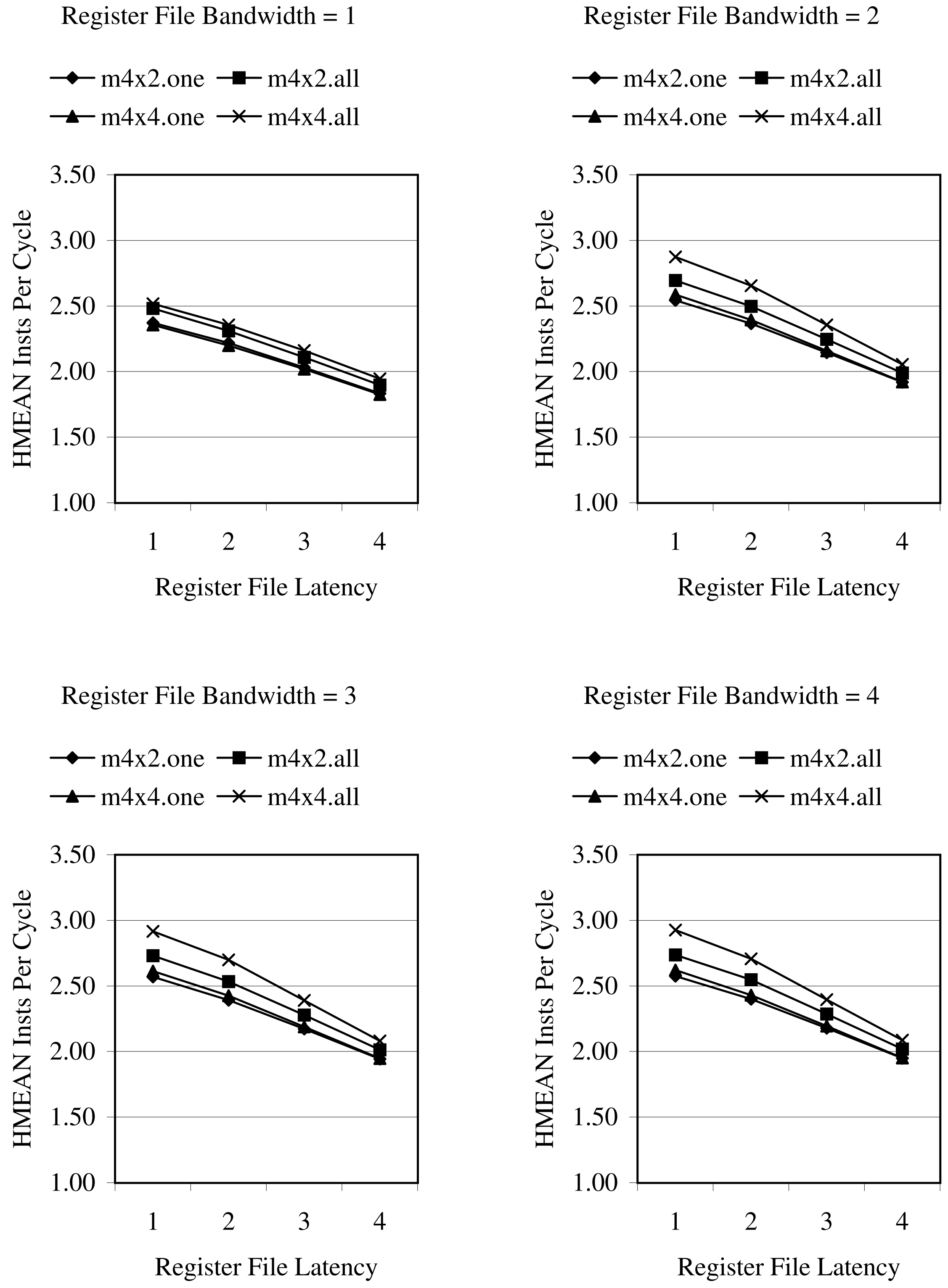


Figure 6.19: SPEC CINT95 performance of 1, 2, 3, and 4 register bandwidth with 1, 2, 3, and 4 cycle register latency for 4 processing units running 2 and 4 tasks, one-at-a-time and all-at-once.

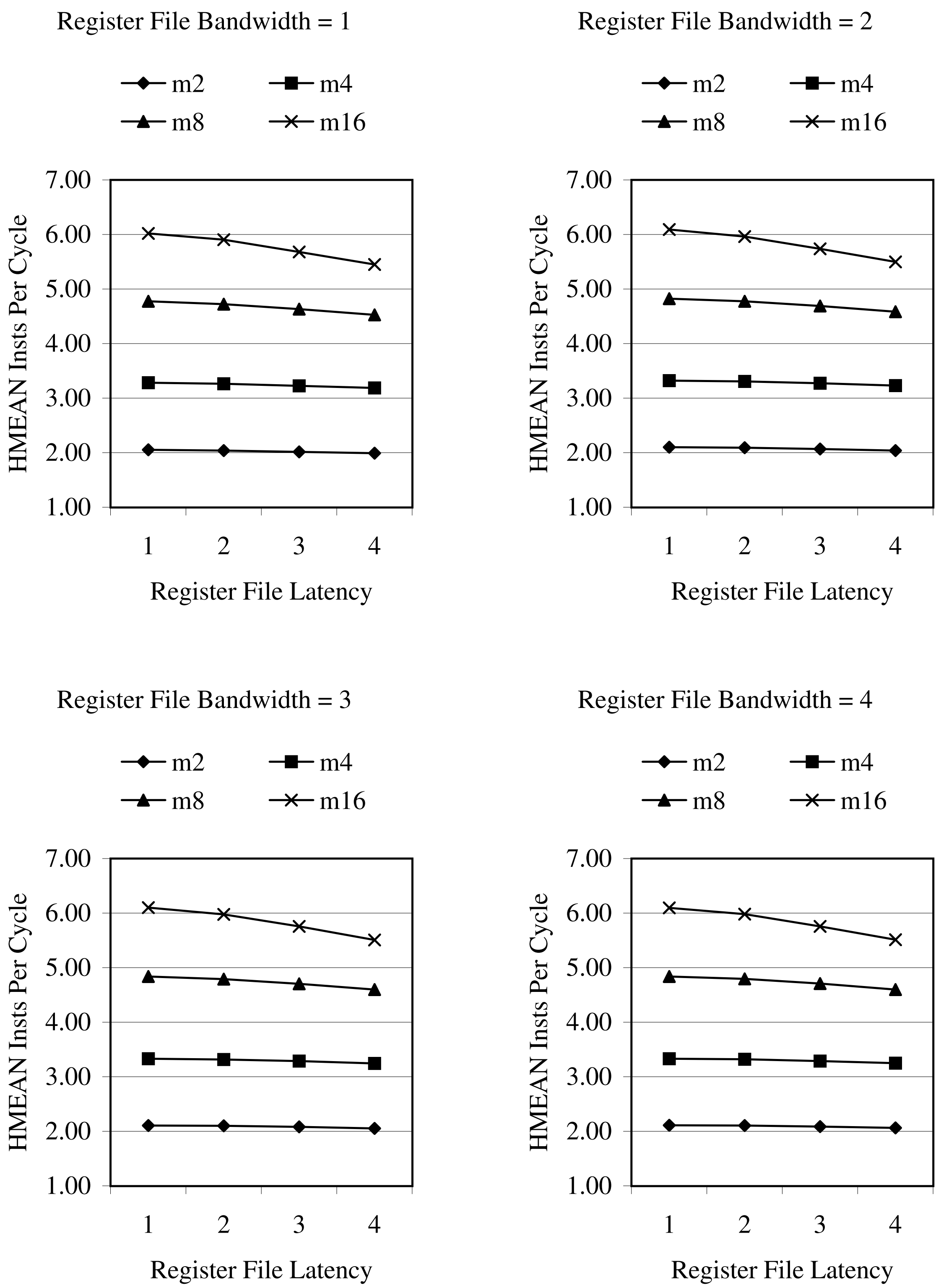


Figure 6.20: SPEC CFP95 performance of 1, 2, 3, and 4 register bandwidth with 1, 2, 3, and 4 cycle register latency for 2, 4, 8, and 16 processing units.

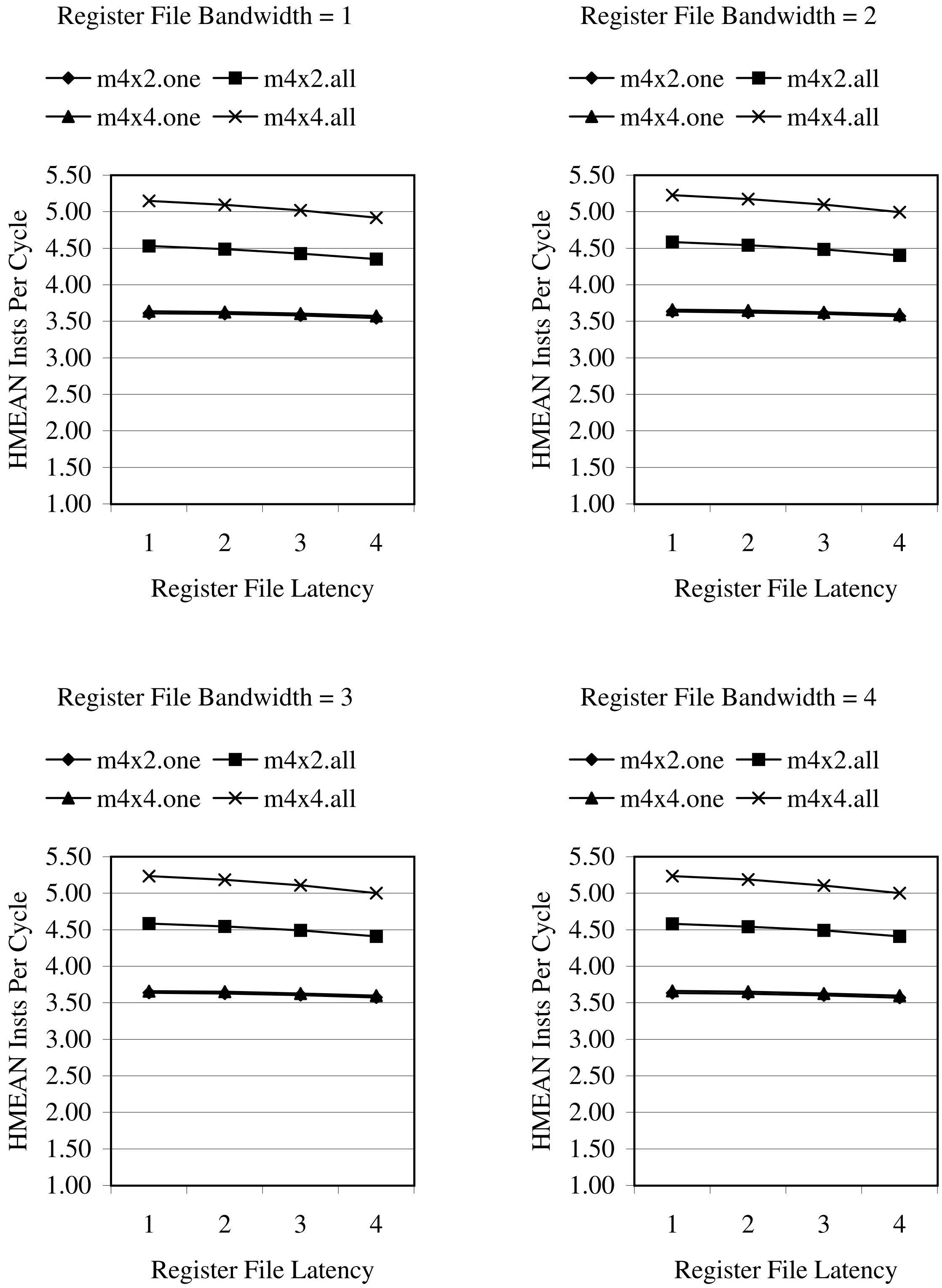


Figure 6.21: SPEC CFP95 performance of 1, 2, 3, and 4 register bandwidth with 1, 2, 3, and 4 cycle register latency for 4 processing units running 2 and 4 tasks, one-at-a-time and all-at-once.

Both aspects must be considered together as each may contribute to a degradation in overall performance. To coincide with these needs, data memory accesses must be allowed to occur speculatively without knowledge of preceding loads or stores. Nevertheless, the corresponding addresses of the loads and stores must be disambiguated dynamically to ensure that true data dependences are honored.

In some respects, the design of the data memory has much in common with the design of the instruction memory (already studied). However, the fact that data memory (unlike instruction memory) may be modified in general and via speculative execution in particular, requires a design of much greater capability. Yet, the design may still rely on adaptations of conventional components for the data cache, but augment them with specialized components, in this case the address resolution buffer and the memory dependence table, to accommodate the characteristics of multiscalar execution.

6.7.1 Data Cache

The key to providing a data memory design that sustains aggressive instruction-level parallel execution is the combination of (i) cache memory, (ii) a high-bandwidth (and low-latency) cache organization, and (iii) a lockup-free cache organization. These ingredients have been studied comprehensively by Sohi and Franklin in the context of aggressive superscalar processors [88]. Yet, the data memory demands in terms of instruction-level parallelism that motivate their use in superscalar processors is equally (if not more) applicable to multiscalar processors. Therefore, this work investigates the use of a cache of similar design to serve as the basis for the data memory of a multiscalar processor.

6.7.1.1 Cache Memory

As described for the instruction memory design, cache memory is used in nearly all microprocessor implementations today to improve performance (for the reasons stated in that description). As also described for the instruction memory design, much work has focused on miss rates as related to overall size, associativity, block size, indexing function, and replacement policy (to name the most well-studied aspects) for caches that provide single access bandwidth. Of course this type of work is relevant to the data memory design, as it is for the instruction memory design. However, again it is necessary to consider additionally the effects of multiple access bandwidth. Moreover, the demands placed on the data memory by a multiscalar processor require further attention to the access latency for the data cache.

Like the instruction cache, the data cache requires a somewhat more involved design, compared to a cache that supports a single access per cycle, because of the need to service multiple accesses per cycle, perhaps many from each processing unit. The use of a high-bandwidth cache organization can provide multiple accesses per cycle, but there are trade-offs in terms of miss rate and access latency depending on the design. (This study does not go into the detail to consider the relationships between access latency and cache size, associativity, *etc*.) Moreover, because the data cache, unlike the task and instruction caches, may perform modifications to the memory in the cache, extra care must be taken in choosing the proper organization (as described below).

6.7.1.2 High-Bandwidth Cache Organizations

High-bandwidth cache organizations are well-developed in the literature and in practice. The common approaches, already described in the instruction memory design, are multi-ported, multi-level, interleaved, and duplicated. According to the order listed, these organizations provide a progression from more centralized, with higher access latency and lower miss rates, to more decentralized, with lower access latency and higher miss rates. As explained for the instruction memory, the multi-ported and multi-level designs are not well suited for use in a multiscalar processor since neither scales well to many ports for the kinds of cache sizes studied here. (Again, the multiscalar processor designs considered might need to scale up to perhaps 16 ports for 16 processing units with total cache sizes up to 128K-byte.)

Though the other alternatives of interleaved and duplicated designs, referred to as shared and private respectively, fit within the design constraints of the instruction memory, the situation is somewhat more involved with respect to the design constraints of the data memory. The reason is that data memory unlike the instruction memory may be modified during execution. (In fact, it may be modified in a speculative manner during execution, but this aspect is handled by the address resolution buffer, described next.) As a result, the use of the private cache organization invites the burden of having to handle the difficulty of coherence among the various private caches. While cache coherence has been extensively investigated in the absence of speculation [6], it has been only recently considered in the presence of speculation [30]. Since a comprehensive investigation of the trade-offs between shared and private cache organizations in the context of speculation is beyond the scope of this work, only the conservative shared cache organization is studied here.

6.7.1.3 Lockup-Free Cache Organizations

The data cache must support concurrency in the handling of both misses as well as hits to sustain an aggressive rate of instruction-level parallel execution. Yet, the canonical implementation of a cache is a lockup cache (though it is usually just referred to as a cache), and it usually supports little or no concurrency among accesses made to it. That is, with a lockup cache, a stall occurs if during the handling of a cache miss, another access is performed on the cache. Such stalls can be avoided by using a lockup-free cache. A lockup-free cache allows the handling of cache hits, cache misses, or both during an outstanding cache miss. The misses handled by a lockup-free cache may be divided into two categories. The first miss to a cache block is called the primary miss. A subsequent miss to the same cache block (during the handling of the miss) is referred to as a secondary miss.

To support multiple in-flight (*i.e.*, outstanding) misses, lockup-free caches require special hardware resources, originally described by Kroft [44]. In Kroft's design, registers called MSHRs (Miss Status Holding Registers) are used to hold information about outstanding misses. The MSHRs save enough information on a miss so that pending accesses for the block can be completed when the fetched cache block arrives from a lower level in the memory hierarchy. One MSHR is associated with each outstanding fetch to a lower level in the memory hierarchy. A primary miss and several secondary misses may be handled by a single fetch request, depending on the design of the MSHR. All lockup-free caches require that information be associated with the returning cache block to match up waiting accesses with the returning cache block contents, unless the contents of the block returns in the order the cache block has been accessed.

The complexity/performance tradeoffs of lockup-free cache designs have been studied by Farkas and Jouppi [22]. In their work, four organizations of MSHRs are described: implicitly addressed, explicitly addressed, in-cache, and inverted. For the use of lockup-free caches in a multiscalar processor, the concern is with ensuring that the capabilities of the MSHRs do not result in the data memory having an undue impact on the overall performance of the processor, not with determining the exact details of their implementation. The impact of a lockup-free cache on processor performance is properly represented as a function of the number and kind of primary misses and secondary misses it supports. The best match among known designs for conventional processors can be used to provide the required capabilities for a multiscalar processor, or novel designs may be investigated as need be.

6.7.2 Address Resolution Buffer

In addition to providing high-bandwidth and low-latency operation that supports concurrency among both hits and/or misses, the data memory must also allow accesses to occur in a speculative manner without knowledge of preceding loads or stores. The corresponding addresses of the loads and stores must be disambiguated dynamically (many in parallel) to ensure that true data dependences are honored. That is, loads and stores must appear to execute in program order, regardless of what actually transpires. The need for this speculation is the limited ability of the compiler to provide the specification and coordination of memory communication in the same way it handles register communication.

Thus, the address resolution buffer, proposed by Franklin and Sohi [25], provides the data memory with speculative storage to complement the architectural storage of the data cache. The basic idea behind the address resolution buffer is to maintain different versions (on a word, data cache block, or other granularity) as a compact unit such that speculative versions are readily discerned from one another and from the architectural version of a particular storage location. Moreover, the address resolution buffer not only provides the speculative versions to allow parallel execution, it also provides the means of disambiguating the loads and stores that are executed to ensure that the correct program semantics are maintained.

6.7.2.1 Structure

The address resolution buffer is similar in organization to a normal cache; it may be direct mapped, set associative, fully associative, *etc.*, where each set contains a number of entries. An entry is the analog of a block in a cache, except that an entry provides storage for a fixed number of versions equal to the number of tasks that may be in concurrent execution. Each entry of the address resolution buffer consists of an address tag, multiple data items (each a word or a cache block in this study), and sets of state bits matched one per data item. Figure 6.22 shows the structure of an address resolution buffer entry.

Address Tag	V	L	S	Data	• • •	V	L	S	Data
		Task 0 Version					*Task N-1 Version*		

Figure 6.22: Structure of an address resolution buffer entry.

The address tag identifies the entry. The state bits consist of a *valid bit* (V) as well as *load and store bits* (L and S) used to determine if the version is valid and if loads and/or stores have been performed on it. The data provides the storage for each version. Any speculative version that is committed or squashed is always removed immediately from the address resolution buffer. This action is a requirement, since speculative and architectural versions do not coexist in the address resolution buffer. Since the storage is centralized, different speculative versions in an entry are readily accessible.

6.7.2.2 Versions and Ordering

It is vital to provide multiple versions of memory storage locations in order to exploit parallelism via speculative execution [8]. These versions are needed to serve as the means to rename memory storage, so that only true dependences guide parallel execution. The amount of storage used per version dictates how much of the memory namespace the address resolution buffer may support (rename). In the original address resolution buffer design, the amount of storage used per version was one word. However, it may be advantageous for the amount of storage used per version to be one (data) cache block. This amount may better coincide with the locality exploited by the data cache and allow the address resolution buffer to support more of the memory namespace with the same number of entries.

Yet, just providing the versions is not enough. In addition, it is necessary to perform ordering among the versions to *disambiguate* loads and stores such that their execution abides by the memory data dependences of the program. The appearance that must be provided for loads and stores to a particular memory address may described as follows: (i) the value read by a load is the value written by the logically preceding store, and (ii) the value written by a store is the value that persists until a logically succeeding store. Conceptually, a load checks the speculative versions of earlier tasks in the address resolution buffer entry to ensure the read condition; a store checks the speculative versions of later tasks in address resolution buffer entry to ensure the write condition.

As described above, the loads and stores of each task may be tracked using the load and store bits provided for each version. However, if load and store bits are provided only at the granularity of a version, then false detections of dependence violations may lead to unnecessary squashes and thereby performance losses (similar to the effects of false sharing in shared memory multiprocessors [20]). In particular, a large version, such as a cache block, is more likely to suffer false detections of dependence violations than a small version, such as a word. In order to mitigate these effects, load and store bits may be maintained at a fine granularity for any amount of version storage using a technique similar to the sector cache [51]. (For the address resolution buffers studied in this work, load and store bits are always provided at a byte granularity.)

6.7.2.3 Operation

When a task executes a load from an address, the address resolution buffer and the data cache are accessed at the same time to determine if both or either contain the address. If the access hits in the address resolution buffer, and the version corresponding to the task is valid, then the data from this version is returned. If the version corresponding to task is invalid, then the closest previous valid version is found by searching the versions of the entry in reverse task order. If no such valid version exists, or the access misses in the address resolution buffer, then data is returned from the data cache. However, if the access also misses in the data cache, then the data is accessed from the next level of the memory hierarchy. If the access missed in the address resolution buffer or the version for the task was invalid, then the load bit in the version is set to enable the detection of potential memory dependence violations.

When a task executes a store to an address, the address resolution buffer and the data cache are accessed at the same time to determine if both or either contain the address. If the access hits in the address resolution buffer, and the version corresponding to the task is valid, then the store is merged with the existing version. If the version corresponding to the task is invalid, then the closest previous valid version is located in the same manner as for a load. The store is merged with this version and entered into the version corresponding to the task. The store bit in the version is set. Moreover, a store to a version invalidates all later versions until the next version with a store bit set, if any, is encountered. An invalidated version that has its load bits set indicates that the corresponding task violated a data dependence by executing a load out of program order with respect to the store. As a result, this task and those that follow it must be squashed, which further requires that all versions corresponding to those tasks be invalidated.

6.7.2.4 Replacement and Writeback

As described above, a load or store causes a miss if no valid corresponding version exists. The data to be supplied for a miss comes from within the bank via the data cache to which the address resolution buffer is paired. Though handling the miss is straightforward, selecting an entry to replace and handling the possible writeback may be involved. The easiest option for an entry to replace is one in which all versions are invalid. If an entry has all invalid versions, it can be replaced with no writeback. The only other option is to replace an entry in which some versions are valid, but this action can only be performed under restricted conditions.

If the task that caused the miss is not the head, then it may be stalled until an entry with all invalid versions becomes available or until it becomes the head (whichever occurs first). If the task that caused the miss is the head, it may writeback versions in the entry associated with it or it may squash other tasks to invalidate other versions of the entry. This approach guarantees forward progress, since a task may always replace entries as the head; and as the head, a task at some point yields to the next task to be the head.

The difficulty with the replacement of entries is caused by the fact that the entries contain speculative updates which cannot be performed as writebacks until the corresponding task is the head. Due to the problematic nature of replacements, the size and associativity along with amount of storage used for a version are important considerations for the address resolution buffer. In particular, the less likely it is that the address resolution buffer uses all of its entries – what is known as an *overflow* – the more likely it is that the address resolution buffer does not unduly impact performance as a result of stalls. Moreover, because all of the writebacks for a task must be performed before the task can commit, the amount of time spent in this operation can likewise influence performance if it delays the committing and assigning of tasks.

6.7.3 Memory Dependence Table

The memory dependence speculation performed using the address resolution buffer is intended to have a positive effect on performance. Nevertheless, if memory dependence speculation is used in an uncontrolled fashion, gains in performance may be eroded by losses due to incorrect execution. On the other hand, controlled memory dependence speculation can avoid this pitfall, but it requires accurate information about the expected characteristics of program execution. In particular, the information that is needed is the identity of dependent producers and consumers of memory data. Because this information is problematic to gather statically via the compiler, the data memory design studied in this work uses an approach that gathers it dynamically during execution.

It is not possible, in general, to determine ahead of time on a per task basis for which memory locations values are consumed or produced. Even in cases where particular memory locations might be identified, the (more or less) unlimited extent of the memory namespace may make specification of locations problematic. Thus, the consumption and production of memory values may not be coordinated (as in the case of register values). However, if a dependence does exist between a store and load in the dynamic task window, and these instructions consistently execute out of program order, then the execution of these instructions may be synchronized so that the load does not execute until after the store.

This concept, proposed by Moshovos, Breach, Vijaykumar, and Sohi [57], is intended to improve the accuracy of memory dependence speculation, while retaining the performance benefits of aggressive speculation. The memory dependence table provides mechanisms that attempt (i) to predict those instructions whose immediate execution is likely to violate a true data dependence and (ii) to delay the execution of those instructions only as long as is necessary to avoid data dependence violations. It does so by building an association between data dependent load-store pairs, based on their history of data dependence violations, and synchronizing their subsequent executions.

6.8 Data Memory Evaluation

The data memory evaluation is divided into two parts in order to focus on the conventional and specialized components separately. Though there is a close connection between the overall performance the data memory affords the processor and the characteristics of all of the data memory components, it is practical to study the components separately since their behaviors are more or less separable from one another. Moreover, this approach is apt because it readily identifies the impact that the components developed for multiscalar execution have on performance with respect to an otherwise conventional (albeit aggressive) data memory.

First, the conventional component study focuses on the extent to which the data cache affects the execution of the processor. Since the specialized components are not the subject of this study, their configuration is fixed so as not to exert undue influence on the results. For the conventional components that are the subject of this study, the evaluation involves varying the data cache size and associativity, for a constant block size and replacement policy. In addition, lockup-free caches are assumed through the study, but their configuration is suitably aggressive, though held constant throughout. Furthermore, the access latency (on a hit) of the data cache is changed to investigate the effect that this characteristic has on the execution of the processor.

Second, the specialized component study focuses on the extent to which the address resolution buffer and the memory dependence table affect the execution of the processor. Similar to the other study, the conventional components in this study are configured so as not to exert undue influence on the results for the specialized components. For the address resolution buffer part of this study, the evaluation involves varying the address resolution buffer size and associativity, as well as changing the amount of storage used for versions between blocks (of the same size as for the data cache) and words. For the memory dependence table part of the study, the evaluation involves using the rest of the data memory with and without the benefit of the memory dependence table. The size, associativity, and other characteristics are suitable to allow the memory dependence table to perform its function effectively, but held constant.

6.8.1 Metrics

Many of the metrics used for the data memory evaluation, particularly those for the data cache study, are the same as those already described for the task cache and instruction cache studies in Section 5.8.1 of the instruction memory evaluation. Since these metrics may be applied to the study of any cache, it is reasonable to use them here for the data memory in the same way as for the instruction memory. Furthermore, rather than restate them all here, the reader is referred to that section on metrics for the instruction memory evaluation in the previous chapter for a general description. A specific difference for the data memory evaluation, though, is that in terms of architectural miss ratio as compared to the speculative miss ratio, the divisor used is the number of useful load/store instructions rather than the number of useful tasks or useful instructions as for the instruction memory evaluation. Furthermore, for effective access latency, only the latency of load instructions is considered since store instructions are performed in the background.

Though the data memory evaluation may make use of the same metrics as the instruction memory evaluation, this convenience only applies to the data cache component of the data memory. The two specialized components of the data memory, the address resolution buffer and the memory dependence table, also have a critical impact on the data memory that must be evaluated. For these specialized components, the metrics of miss ratio and effective access latency are either not informative or not applicable to their study. Because the address resolution buffer and the memory dependence table strike directly at the performance the processor may attain using the data memory, the best metric for their study is the actual effect that each has on the overall performance of the processor. Thus, in this evaluation of the data memory, the useful instructions per cycle, given as the harmonic mean (HMEAN) for the SPEC CINT95 and SPEC CFP95 benchmarks (described in Section 4.6.1), is used for the study of the address resolution buffer and memory dependence table.

6.8.2 Conventional Component Study

For the conventional component study, the specialized components are fixed to the standard configuration. Likewise, because the data memory is the subject of this evaluation, the other components of a multiscalar processor are also fixed to the standard configuration shown in Table 3.11 of Chapter 3. As indicated above, the conventional component under consideration is the data cache; the configurations used in its study are described below.

6.8.2.1 Configurations

The data cache is configured only as a shared organization (due to complications in maintaining cache coherence for a private organization, already discussed). The data cache size is varied among 16K-byte, 32K-byte, 64K-byte, and 128K-byte overall for all numbers of processing units (with the storage divided equally in a like number of banks). The block size (32 bytes) and the replacement policy (least recently used) are fixed in this evaluation, but the associativity is varied among 1-way, 2-way, and 4-way. In addition, the data cache is configured with different access times (on a hit) of 1, 2, 3, and 4 cycles (pipelined where the pipeline is flushed on a miss) to study this effect (due to the interconnect) on overall processor performance. Moreover, a range of processing unit configurations are used. In particular, the number of processing units is varied among 2, 4, 8, and 16, as well as among 4 processing units running 2 and 4 tasks one-at-a-time and all-at-once. Using these configurations, the data cache results are gathered in two steps.

First, the data cache evaluation focuses in general on the impact of different sizes among the processing unit configurations. For this evaluation, only 2-way associative data caches are used (the results for the other associativities are similar, though shifted higher or lower as seen in the second part of the evaluation). Only processing unit configurations that run a single task per processing unit are discussed. The overall differences among configurations are not very significant, and the multiple tasks per processing unit are similar to the results for the single task per processing unit configurations. In addition, this part of the data cache evaluation focuses on the behavior of one configuration, 64K-byte with 2-way associativity, for different access (on a hit) latencies. Second, the data cache evaluation focuses in specific detail on the behavior with different sizes and associativities for only one processing unit configuration, 16 processing units. The relative differences for all metrics with changing cache sizes and associativities is the same across processing unit configurations.

6.8.2.2 General Results

Figure 6.23 gives the speculative (on the left) and architectural (on the right) miss ratios with different shared data cache sizes and 2-way associativity for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. For the integer benchmarks, the speculative miss ratio, misses per access, is small in absolute terms for all shared data cache sizes, not more than 0.02 misses per access. However, in relative terms, there is a large difference between the smallest (16K-byte) and the largest (128K-byte) shared data cache organizations. In particular, the misses per access are 3 to 4 times greater for smaller as compared to larger shared data caches depending on the processing unit configuration.

Among the different processing unit configurations, the speculative miss ratios do not change significantly, not by much more that 0.005 misses per access. Nonetheless, the trends indicate that configurations with more processing units have lower miss ratios than those with less processing units. This behavior is partly due to the fact that speculative execution may have a prefetch effect (as described in the instruction memory evaluation) and partly due to the effect of combining that occurs for the multiple banks of the shared cache (as also described in the instruction memory evaluation). Though the prefetching effect is difficult to quantify, the combining effect is not. Of the accesses performed on the data caches, the amount of combining ranges from around 6% of accesses for 2 processing unit configurations to 3% of accesses for 16 processing unit configurations. In contrast to the instruction memory results which show that combining increases as the number of processing units (and banks) increase, the data memory results show that combining instead decreases in this situation. This observation likely indicates that though the same instructions execute on different processing units, these instructions operate on different data.

As already stated, the combining effect may be measured more easily than the speculative prefetching effect. Yet, even though the combining may be measured, it is still difficult to separate the two effects, since some of the combining occurs for speculative accesses that are not useful. However, some of these speculative accesses likely ensure that blocks remain in the cache for later accesses that are useful. In order to help understand the nature of this behavior the architectural miss ratios may be of some use. For the integer benchmarks, the overall miss ratios are slightly higher for architectural as compared to the speculative. Moreover, the results show that the architectural miss ratios, the misses per useful load/store are about the same regardless of the processing unit configuration (the plots in the graph are all on top of one another). Recall, different numbers of processing units are capable of performing different amounts of speculation, more speculation with more processing units and less speculation with less processing units. It appears to be the case that the lower speculative miss ratio for more processing units is attributable to otherwise non-useful instructions since this miss ratio advantage is not evident for the architectural miss ratios. Nonetheless, there does appear to be a balance in terms of the amount of speculation and its utility since the architectural miss ratio is not higher for more processing units than for less.

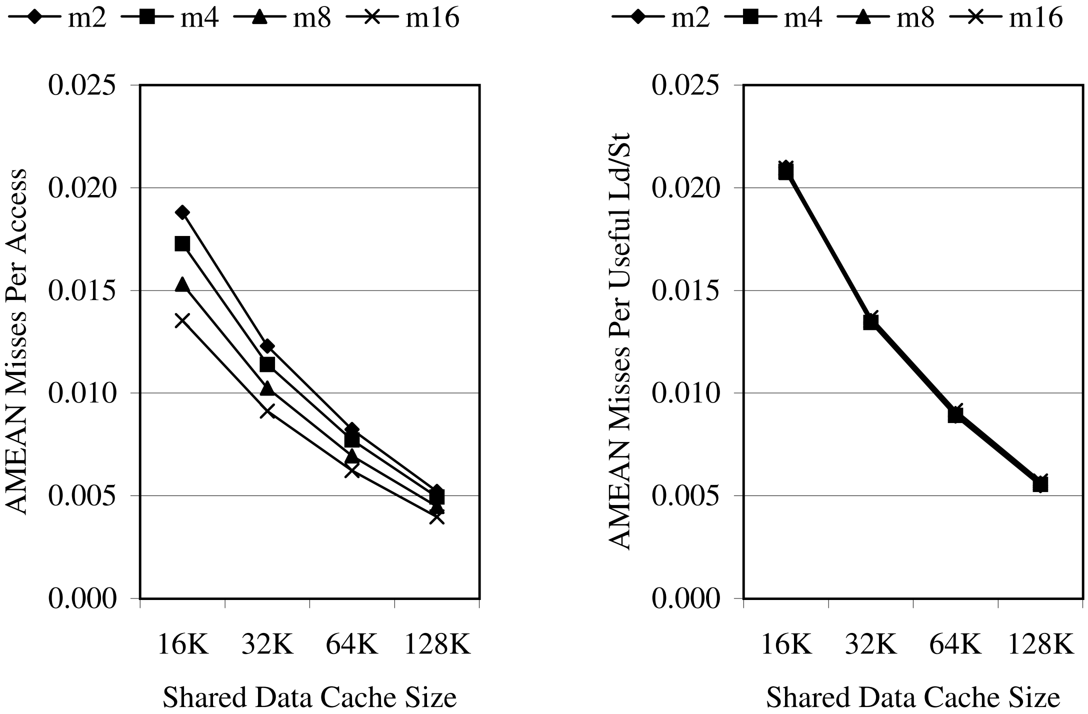


Figure 6.23: SPEC CINT95 misses per access and misses per useful load/store with 2-way associative shared data caches for 2, 4, 8, and 16 processing units.

Figure 6.24 gives the speculative (on the left) and architectural (on the right) miss ratios with different shared data cache sizes and 2-way associativity for the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. For the floating benchmarks, the speculative miss ratio, misses per access, is somewhat large in absolute terms for all shared data cache sizes, more than 0.03 misses per access, which is 50% higher than for the integer benchmarks. However, in relative terms, there is not as large a difference between the smallest (16K-byte) and the largest (128K-byte) shared data cache organizations as for the integer benchmarks. The misses per access are less than half as much for smaller as compared to larger shared data caches across the processing unit configurations.

Among the different processing unit configurations, the speculative miss ratios do not change significantly, not by much more that 0.006 misses per access. This amount for the floating point benchmarks is similar to the amount for the integer benchmarks, though from the scale of the graph it may not appear to be. Furthermore, the trends indicate that, as was the case for the integer benchmarks, configurations with more processing units have lower miss ratios than those with less processing units, for the floating point benchmarks. Likewise, the reasons for this behavior, partly due to the prefetch effect of speculative execution and partly due to the combining for the multiple banks of the shared cache, are the same in the floating point case as in the integer case. As for the impact of combining (the impact of the prefetching effect is difficult to quantify as already described), the amount of combining ranges from around 10% of accesses for 2 processing unit configurations to 14% of accesses

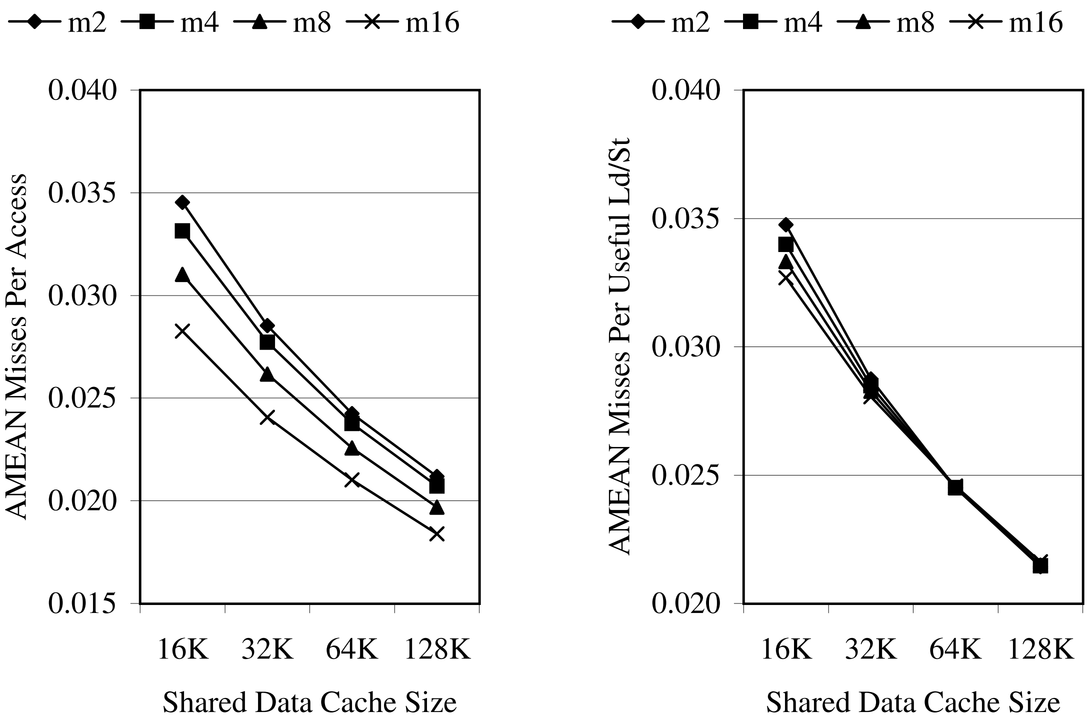


Figure 6.24: SPEC CFP95 misses per access and misses per useful load/store with 2-way associative shared data caches for 2, 4, 8, and 16 processing units.

for 16 processing unit configurations. In contrast to the integer results which show that combining decreases as the number of processing units (and banks) increase, the floating point results show that combining instead increases in this situation. Thus, in the floating point case, there seems to be more spatial cache locality than in the integer case.

As for the integer benchmarks, the architectural miss ratios may be used to help understand the nature of the combining and prefetching behavior for the floating point benchmarks as well. For the floating point benchmarks, the architectural miss ratios are slightly higher as compared to the speculative miss ratios, similar to the integer benchmarks. Likewise, as for the integer benchmarks, the results show that for the floating point benchmarks, the architectural miss ratios, the misses per useful load/store, are about the same regardless of the processing unit configuration (the plots in the graph are all on top of one another). However, the architectural miss ratios for the most aggressive processor configuration of 16 processing units do stand out as being slightly higher than those of other less aggressive processor configurations. Nevertheless, it may be the case for the floating point benchmarks, as for the integer benchmarks, that the lower speculative miss ratio for more processing units is attributable to otherwise non-useful instructions since this miss ratio advantage is not evident for the architectural miss ratios. However, because control speculation accuracy is higher for floating point than for integer benchmarks there may be an alternative explanation. In particular, it may be the case that because the overall miss ratios are higher for floating point as compared to integer programs, that any non-useful miss has a more profound effect since it consumes resources that are already in short supply.

Figure 6.25 gives the effective load latency (on the left) and performance (on the right) with different shared data cache sizes and 2-way associativity for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Just beneath, Figure 6.26 gives the performance of a fixed 64K-byte 2-way associativity shared data cache for access (on a hit) latencies of 1, 2, 3, and 4 cycles on the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. In addition to these results for the integer benchmarks, the same results are given for the floating point benchmarks. Figure 6.27 gives the effective load latency (on the left) and performance (on the right) with different shared data cache sizes and 2-way associativity for the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Just beneath, Figure 6.28 gives the performance of a fixed 64K-byte 2-way associativity shared data cache for access (on a hit) latencies of 1, 2, 3, and 4 cycles on the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Though the absolute numbers for the two groups of benchmarks are obviously quantitatively different, it is interesting to compare them qualitatively.

Overall, the trends in terms of performance, useful instructions per cycle, are about the same for the integer and floating point benchmarks. As the data cache size is doubled there is a slight but steady increase in performance for both the integer and floating point benchmarks, though rate is somewhat more for integer than for floating point. The reason for the differences in rate is likely due to the differences in effective load latency for the two groups of benchmarks. The effective load latency decreases as the cache size increases and may in fact even go below the hit latency due to the combining of accesses to the same cache block that may occur. Nonetheless, the characteristics of the effective load latency are much different for integer and floating point benchmarks with different processing unit configurations as well as overall. With respect to different processing unit configurations, the effective load latency is influenced much by the number of processing units for floating point benchmarks but not much at all for the integer benchmarks. All of the processing unit configurations have about the same effective load latency for the integer benchmarks. However, the effective load latency is increased by as much as a cycle each time the number of processing units is doubled for the floating point benchmarks.

The reason for this behavior is that the demand for memory bandwidth is considerably higher for the floating point benchmarks than for the integer benchmarks, so there is considerably more contention at the cache banks. Moreover, since not all of the accesses go to the same cache block, cache bank conflicts which cannot be accommodated by cache block combining occur, and the accesses involved are stalled. Thus, the demand for memory bandwidth coupled with the amount of parallelism generates contention in the data memory that causes the latency to grow. However, the higher levels of parallelism also afford a greater tolerance for the latency, so performance does not suffer as much as one might think. Moreover, this behavior is reflected in the results that show the effect of changing the load hit latency. Though changes to the load hit latency clearly affect the performance on both groups of benchmarks, the performance for the integer benchmarks appears to decline more rapidly than does the performance for the floating point benchmarks. Because there is higher parallelism in the floating point benchmarks, there is more tolerance for the increases in load hit latency than for the integer benchmarks.

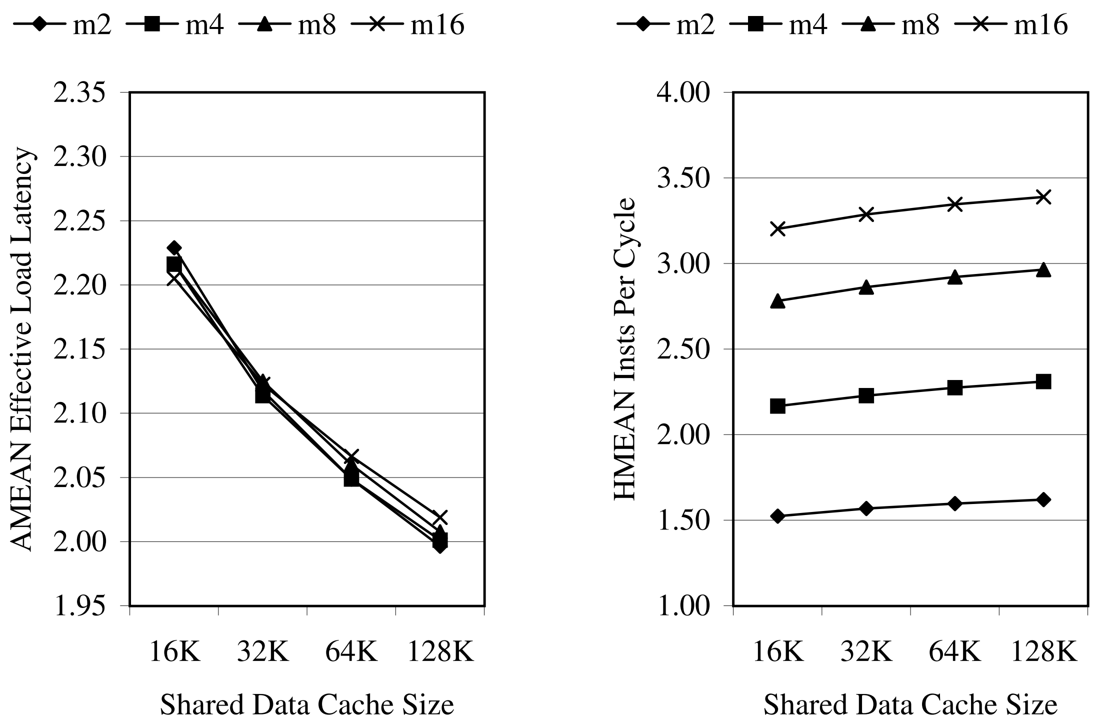


Figure 6.25: SPEC CINT95 effective load latency and performance with 2-way associative shared data caches for 2, 4, 8, and 16 processing units.

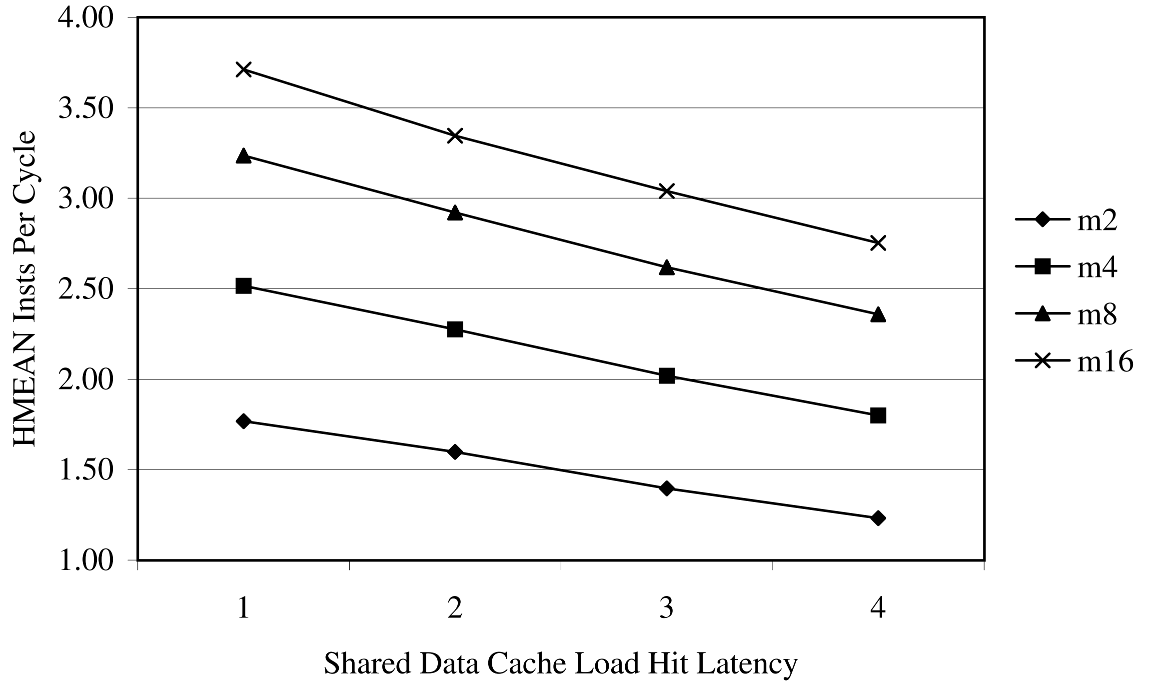


Figure 6.26: SPEC CINT95 performance using 64K-byte 2-way associative shared data cache for load hit latencies of 1, 2, 3, and 4 cycles.

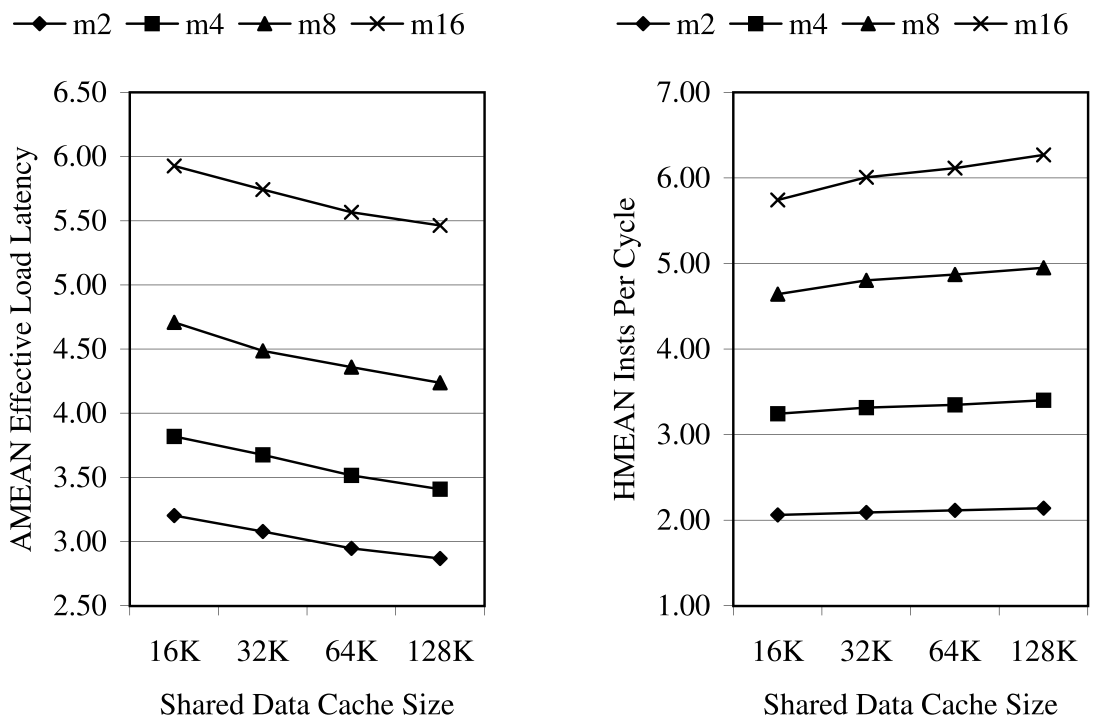


Figure 6.27: SPEC CFP95 effective load latency and performance with 2-way associative shared data caches for 2, 4, 8, and 16 processing units.

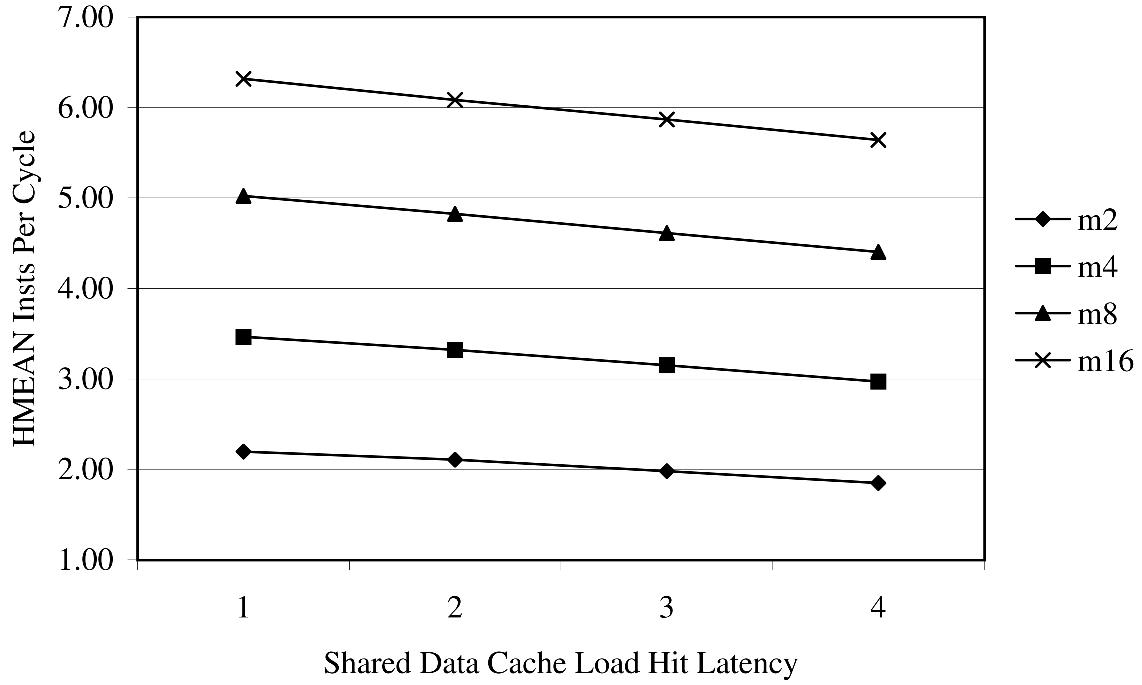


Figure 6.28: SPEC CFP95 performance using 64K-byte 2-way associative shared data cache for load hit latencies of 1, 2, 3, and 4 cycles.

6.8.2.3 Specific Results

Figure 6.29 gives the speculative (on the left) and architectural (on the right) miss ratios with different shared data cache sizes and associativities for the SPEC CINT95 benchmarks using the 16 processing unit configuration. The speculative miss ratio, misses per access, decreases steadily as the data cache size is doubled for all associativities. Among the different associativities, there is a more significant difference going from 1-way to 2-way associativity (sometimes as much as 0.004 misses per access) and a less significant difference going from 2-way to 4-way associativity (usually around 0.001 misses per access). Overall, though the differences in miss ratios are fairly limited. The associativity becomes less of a factor as the data cache size increases, but higher associativity consistently gives lower speculative miss ratios. In absolute terms, the miss ratio for the smallest (16K-byte) and the largest (128K-byte) data caches differ by not much more than 0.01 misses per access. Yet, in relative terms these differences give a miss ratio that as much as 3 times higher for the smallest as compared to the largest data caches. The absolute and the relative differences for the architectural ratios are the same (since only the divisors are different) as for the speculative ratios. What is important to observe is that the misses per useful load/store is significantly higher than the number of misses per access for this aggressive processing unit configuration. Thus, a significant fraction of the misses supported by the data cache in this case are not a result of accesses initiated for the execution of a useful load/store.

Figure 6.30 gives the effective load latency (on the left) and the performance (on the right) with different shared data cache sizes and associativities for the SPEC CINT95 benchmarks using the 16 processing unit configuration. In accordance with the behavior of the miss ratios, the effective load latency decreases steadily as the data cache size is doubled for all associativities. Among the different associativities, there is a more significant difference going from 1-way to 2-way associativity (not more than 0.1 cycle) and a much less significant difference going from 2-way to 4-way associativity. The associativity becomes less of an influence as the data cache size increases, but higher associativity consistently gives lower effective load latency. Overall, the effective load latency for the smallest (16K-byte) and the largest (128K-byte) data caches never differs by more than 0.3 cycles per access, a modest difference of a little more than 10% in effective load latency. The difference in effective load latency has a direct, corresponding effect in terms of performance. In fact, the shapes of plots just seem to be reflections of one another based on the metric being considered. The performance difference in instructions per cycle for the smallest and largest data cache sizes is less than 0.25 instructions per cycle for the 1-way associative designs and around 0.10 instructions per cycle for 2-way and 4-way associative designs.

Figure 6.31 gives the speculative (on the left) and architectural (on the right) miss ratios with different shared data cache sizes and associativities for the SPEC CFP95 benchmarks using the 16 processing unit configuration. The basic data cache behavior with respect to the miss ratios is similar for the floating point as for the integer benchmarks. Nevertheless, there are some differences. Though, the miss ratio does decrease with increasing cache size and associativity, as for the integer benchmarks, the shape of the curves are somewhat distorted for the floating benchmarks for the different associativities. In particular, the associativity has no effect on the miss ratio for the smallest (16K-byte) data cache, and furthermore does

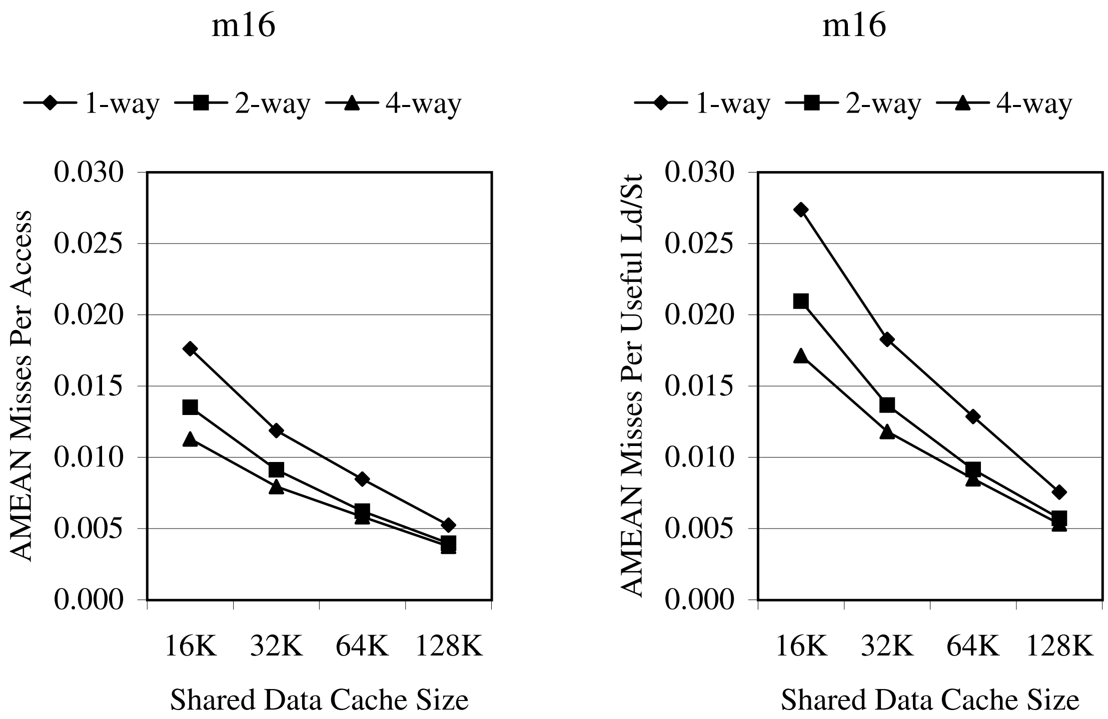


Figure 6.29: SPEC CINT95 misses per access and misses per useful load/store with 1-way, 2-way, and 4-way associative shared data caches for 16 processing units.

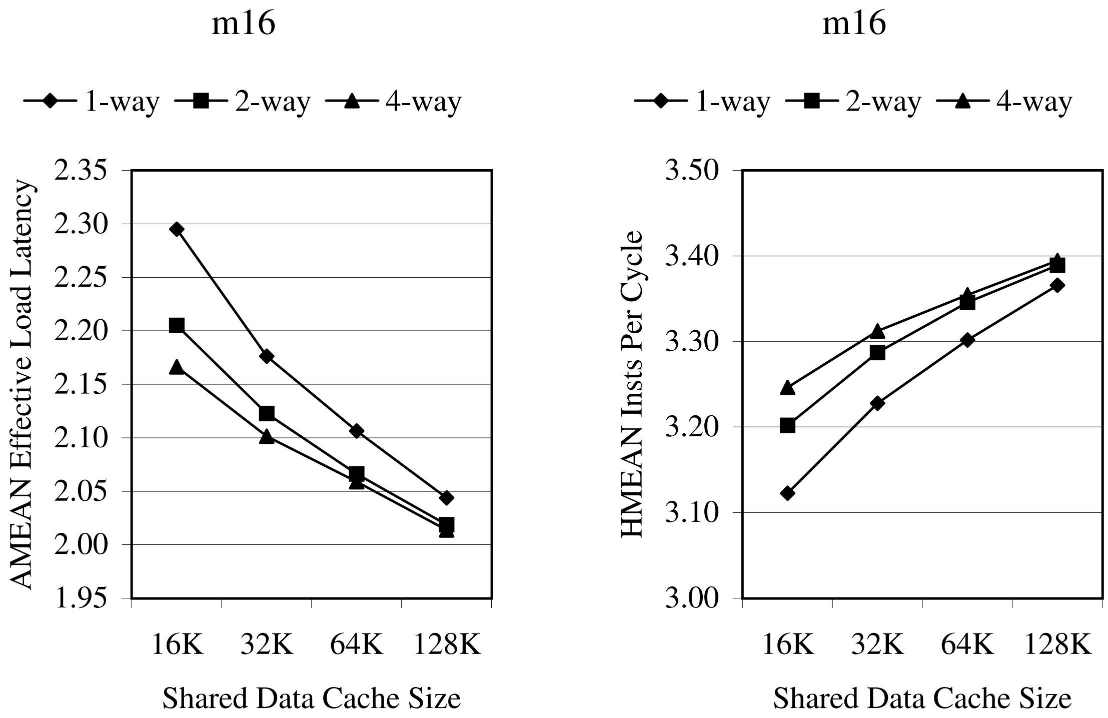


Figure 6.30: SPEC CINT95 effective load latency and performance with 1-way, 2-way, and 4-way associative shared data caches for 16 processing units.

not decrease uniformly as the cache size is doubled (especially for the 2-way associativity configuration). In comparing the speculative and the architectural miss ratios, the same basic behavior is apparent. However, the architectural miss ratios are higher than the speculative miss ratios for the floating point benchmarks, just as for the integer benchmarks. Moreover, the relative differences between the speculative and architectural miss ratios is slightly higher for the different associativities.

Figure 6.32 gives the effective load latency (on the left) and the performance (on the right) with different data cache sizes and associativities for the SPEC CFP95 benchmarks using the 16 processing unit configuration. The floating point benchmark behavior trends are similar to those observed for the integer benchmarks, though the shapes of curves reflect the differences in the miss ratio characteristics between the two. However, the magnitude of the difference in effective load latency as well as in performance is quite a bit larger for the SPEC CFP95 than for the SPEC CINT95. Among the different associativities, there is a bigger difference in going from 1-way to 2-way associativity (around 0.2 cycles per access) and a smaller difference in going from 2-way to 4-way associativity (much less than 0.1 cycles per access). For the most part, the associativity becomes less of an influence as the data cache size increases for floating point benchmarks, as for the integer benchmarks. Overall, the effective load latency of the smallest (16K-byte) and the largest (128K-byte) data caches differs by as much as 0.3 cycles per access, a much more significant difference for the floating point as compared to the integer benchmarks in absolute terms, though not in relative terms. Accordingly, the performance difference in instructions per cycle for the floating point benchmarks between the smallest and largest cache sizes is as much as 0.50 instructions per cycle, nearly 2 times the difference for the integer benchmarks.

6.8.2.4 Real versus Ideal

In order to put the results of this data cache (conventional component) study in perspective, it is worthwhile to consider whether further improvements in data cache operation (aside from changing the hit latency) might provide further improvements in overall processor performance. An appropriate means of making such a determination is to compare the difference in overall processor performance using a real data cache (like the ones already studied) and using an ideal data cache.

An ideal data cache has the same hit latency as a real data cache. However, unlike a real data cache which obviously has finite resources, it has infinite storage capacity (never has a miss), and it has infinite bandwidth (never has an address or a data conflict). The address resolution buffer and memory dependence table (specialized components) used in this comparison are not ideal, but their configuration is such that it does not influence the results.

Figure 6.33 compares the performance on the SPEC CINT95 benchmarks of real 2-way associative shared data caches with 16K-byte, 32K-byte, 64K-byte, or 128K-byte of banked storage to that of an ideal data cache for processor configurations of 2, 4, 8, or 16 processing units running a single task per processing unit (one-at-a-time). Just beneath this figure, Figure 6.34 provides the same comparison on the SPEC CFP95 benchmarks.

The integer benchmarks show small performance increases up to the biggest real data

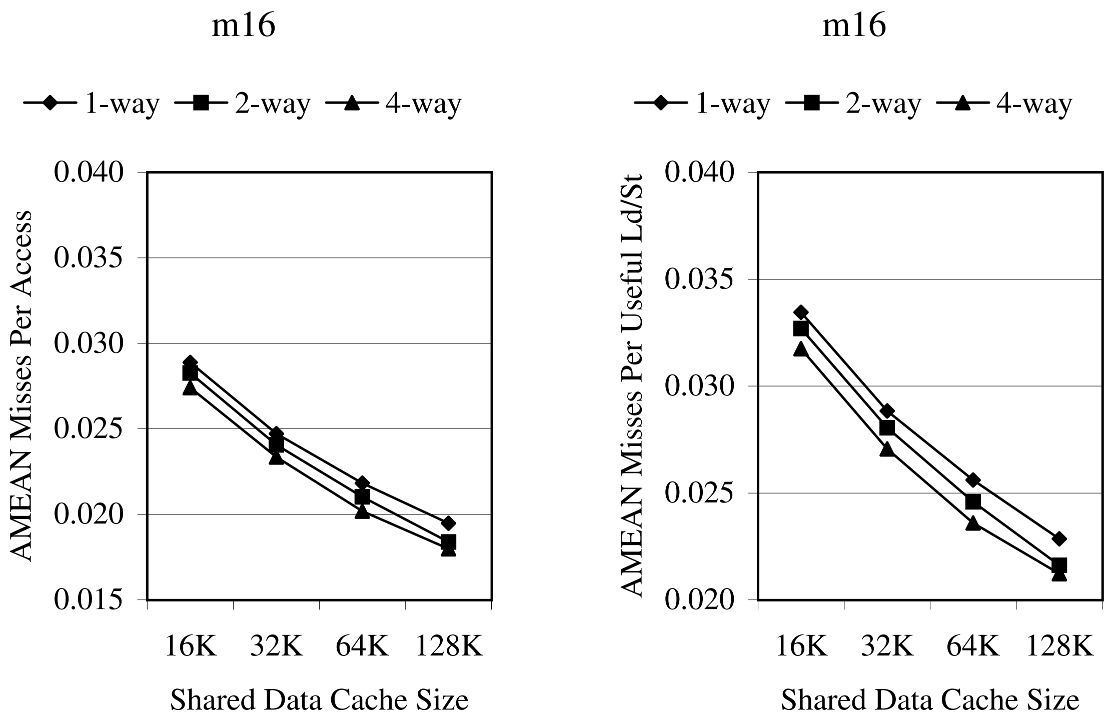


Figure 6.31: SPEC CFP95 misses per access and misses per useful load/store with 1-way, 2-way, and 4-way associative shared data caches for 16 processing units.

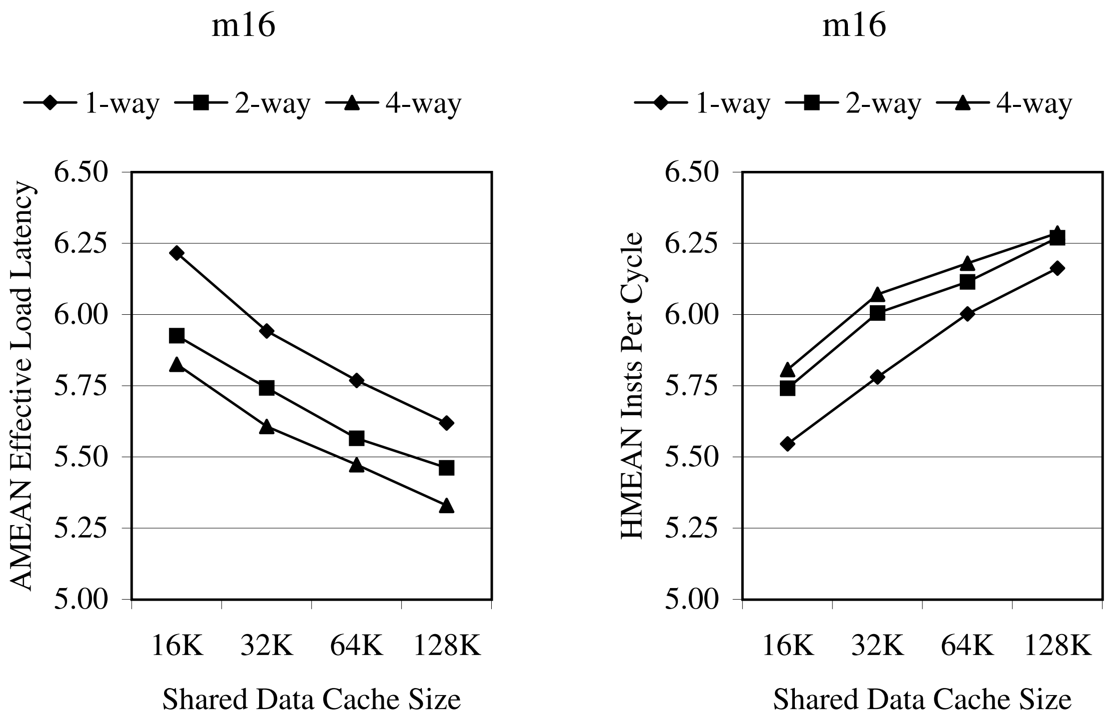


Figure 6.32: SPEC CFP95 effective load latency and performance with 1-way, 2-way, and 4-way associative shared data caches for 16 processing units.

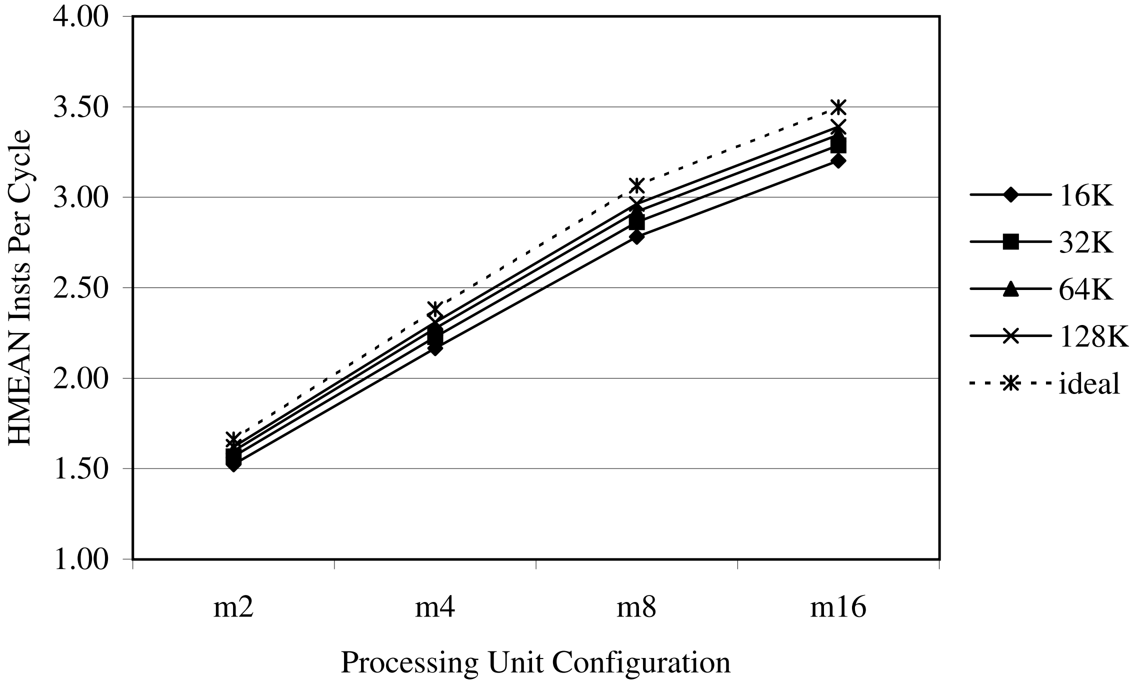


Figure 6.33: SPEC CINT95 performance of real 2-way associative shared compared to ideal data caches for 2, 4, 8, and 16 processing units.

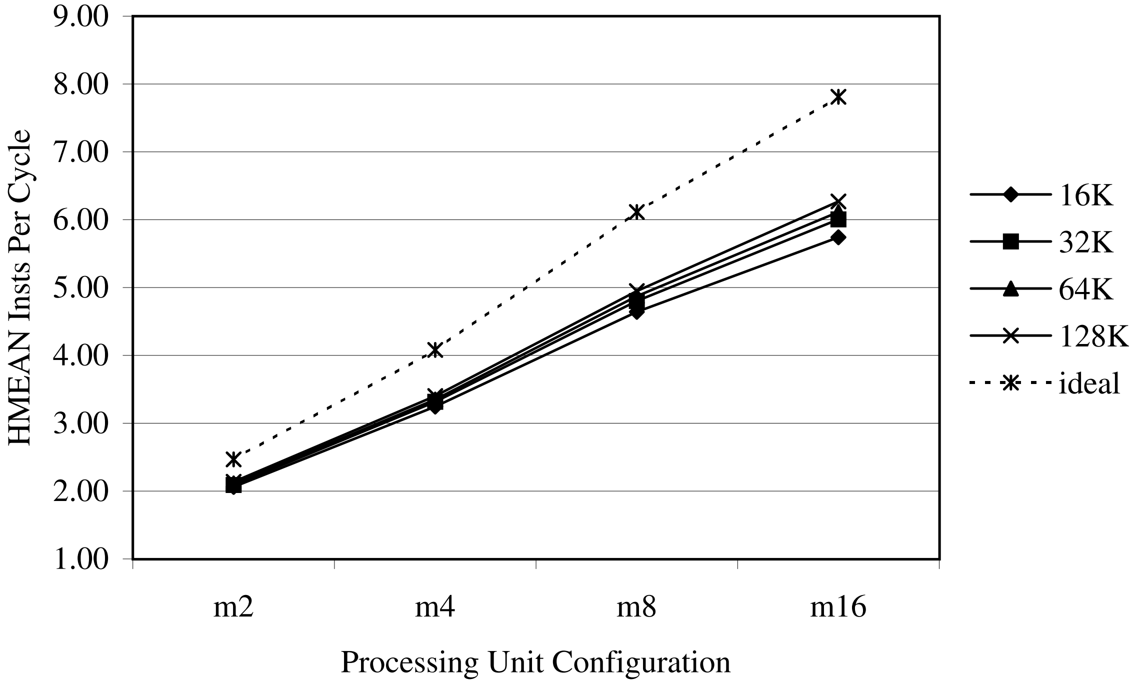


Figure 6.34: SPEC CFP95 performance of real 2-way associative shared compared to ideal data caches for 2, 4, 8, and 16 processing units.

cache considered. However, these improvements as the data cache size is doubled, which are small to begin with, are almost imperceptible for all but the smallest data cache size. Nonetheless, the ideal data cache offers a clear, though modest, performance advantage over even the biggest real data cache. This improvement, though, is due to the infinite bandwidth rather than the infinite storage aspect of the ideal data cache, as miss rates are low and conflicts are high, relatively speaking.

The floating point benchmarks show almost imperceptible performance increase up to the biggest real data cache considered. However, the ideal data cache offers a huge performance advantage, far more significant than for any other ideal cache considered in this thesis, over even the biggest real data cache. The infinite storage aspect of the ideal data cache plays a part in this increase, since the miss rate are relatively high compared to the other cache types examined in this thesis. Yet, most of this increase is due to the infinite bandwidth aspect of the ideal data cache, which reduces the hit latency of the data cache, especially when more processing unit are used, due to data conflicts at the banks.

6.8.3 Specialized Component Study

The specialized components under consideration for this study are the address resolution buffer (ARB) and the memory dependence table (MDT). The details of their configuration, in particular with regard to one another, are supplied below. In addition, the configuration of the data cache, the conventional component of the data memory, is described as it relates to this part of the data memory evaluation. As in the other studies performed in this work, the remaining components of a multiscalar processor, are held constant in the standard configuration as given in Table 3.11 of Chapter 3.

6.8.3.1 Configurations

The data cache for the study of the specialized components is fixed in a shared organization with as many banks as processing units. The overall size is 64K-byte with 2-way associativity using least recently used replacement. The parameters varied for this study only involve the specialized components. However, to isolate the address resolution buffer from the memory dependence table and vice versa in their respective studies, the memory dependence table is fixed to the standard configuration for the address resolution buffer results, and the address resolution buffer is fixed for the memory dependence table results.

For the address resolution buffer results, the size per bank is varied among 8, 16, and 32 entries for all numbers of processing units, while the associativity is varied as powers of two from direct mapped to fully associative for each of the address resolution buffer sizes. Note, these configurations are per bank, and that disambiguation of load/store instructions is at a byte granularity. Moreover, the amount of storage used for the versions of an entry is varied between a cache block (of the same size as for the data cache) and a word. The cache block design is referred to as having block entries, and the word design is referred to as having word entries. For the memory dependence table results, the size is fixed at 16 entries per bank with full associativity for runs when it is used and at 0 entries when it is not used, for all numbers of processing units.

The address resolution buffer results focus on the impact of different address resolution buffer sizes and associativities among the processing unit configurations. For this evaluation, only processing unit configurations that run a single task per processing unit are discussed. The overall differences among configurations that use multiple tasks per processing unit are similar to the results for those that use a single task per processing unit. The address resolution buffer results also focus on the impact of the size of the unit of entry storage to illustrate the effect that this characteristic can have on performance in conjunction with the number of entries and their associativity. The memory dependence table results only focus on the performance impact with or without its use, instead of the specifics of its organization.

6.8.3.2 Address Resolution Buffer Results

Figure 6.35 gives the performance of direct mapped to fully associative address resolution buffers with 8 word (on the left) or block (on the right) entries for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Figures 6.37 and 6.39 give the same information for address resolution buffers of 16 and 32 word (on the left) and block (on the right) version entries, respectively. For the integer benchmarks, all word entry address resolution buffers steadily provide similar increases in performance for all processing unit configurations as the associativity increases. Moreover, the rate for the performance increases is directly correlated to the aggressiveness in number of processing units of the processor configuration. For instance, the increase for 2 processing units going from 1-way to 8-way associativity is only about 0.10 instructions per cycle, while the increase for 16 processing units is nearly 1.50 instructions per cycle. The reason for this behavior is that the smaller configuration places less demand on the address resolution buffer than the larger configuration does; so, the larger configuration benefits more from the flexibility of higher associativity (in terms of stalls) than the smaller configuration. However, there seems to be a knee above 8-way associativity where there is no apparent benefit in further associativity increases. In contrast to the word entry results, the block entry results show only a modest improvement from increases in associativity. In addition, this improvement diminishes (to the point of being practically non-existent) as the number of block entries is increased.

Figure 6.36 gives the performance of direct mapped to fully associative address resolution buffers with 8 word (on the left) or block (on the right) entries for the SPEC CFP95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Figures 6.38 and 6.40 give the same information for address resolution buffers of 16 and 32 word (on the left) and block (on the right) entries, respectively. For the floating point benchmarks the performance is more sensitive to the address resolution buffer than for the integer benchmarks, for word as well as block entries. With respect to word entry configurations, the performance always appears to be suppressed (never leveling off, and implying the need for more entries) for all of number of entries studied. Though the performance is suppressed for all sizes, it is particularly so for associativities below 4-way; for associativities of 4-way and above, the rate of performance improvement seems to accelerate rapidly. It seems clear that the word entry design simply cannot provide enough storage, at any associativity, to satisfy the needs of the floating point benchmarks. However, this effect is not limited to the word entry address resolution buffers. It also occurs for the block entry address resolution buffers, which can support quite a bit

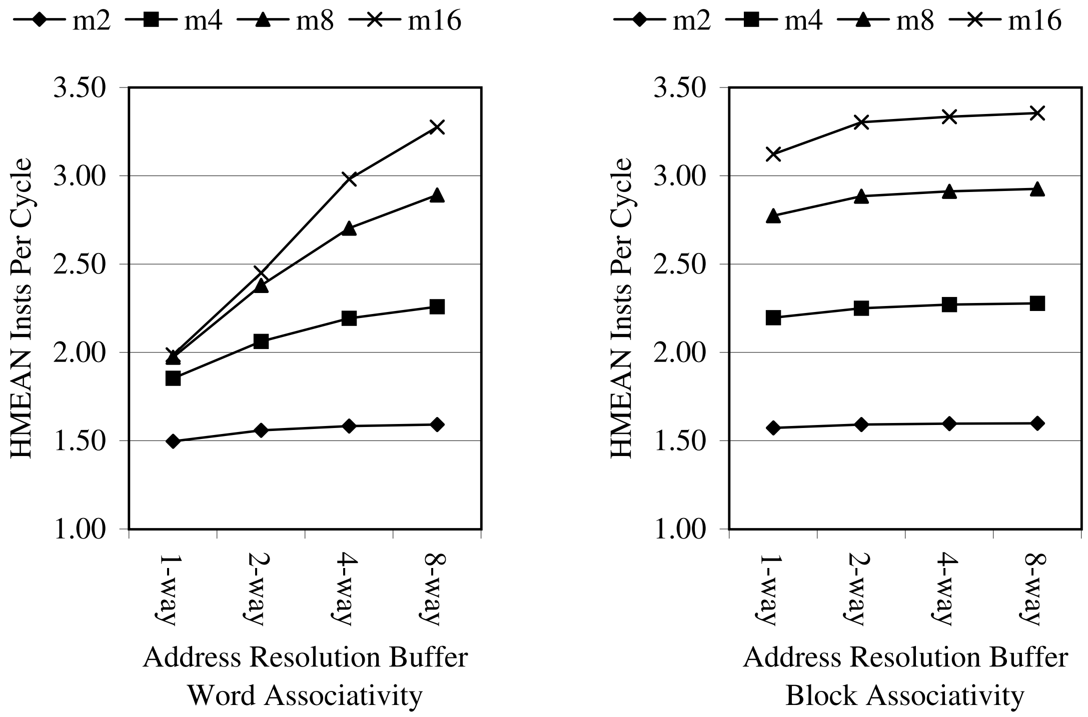


Figure 6.35: SPEC CINT95 performance of direct mapped to fully associative address resolution buffers with 8 word or block entries for 2, 4, 8, and 16 processing units.

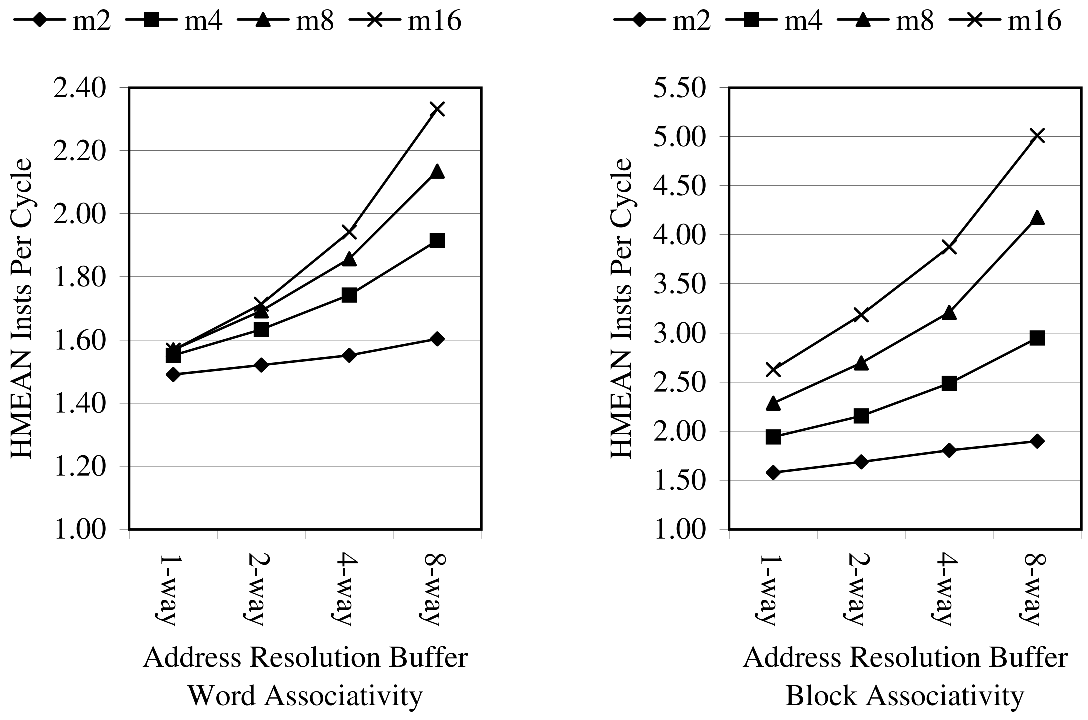


Figure 6.36: SPEC CFP95 performance of direct mapped to fully associative address resolution buffers with 8 word or block entries for 2, 4, 8, and 16 processing units.

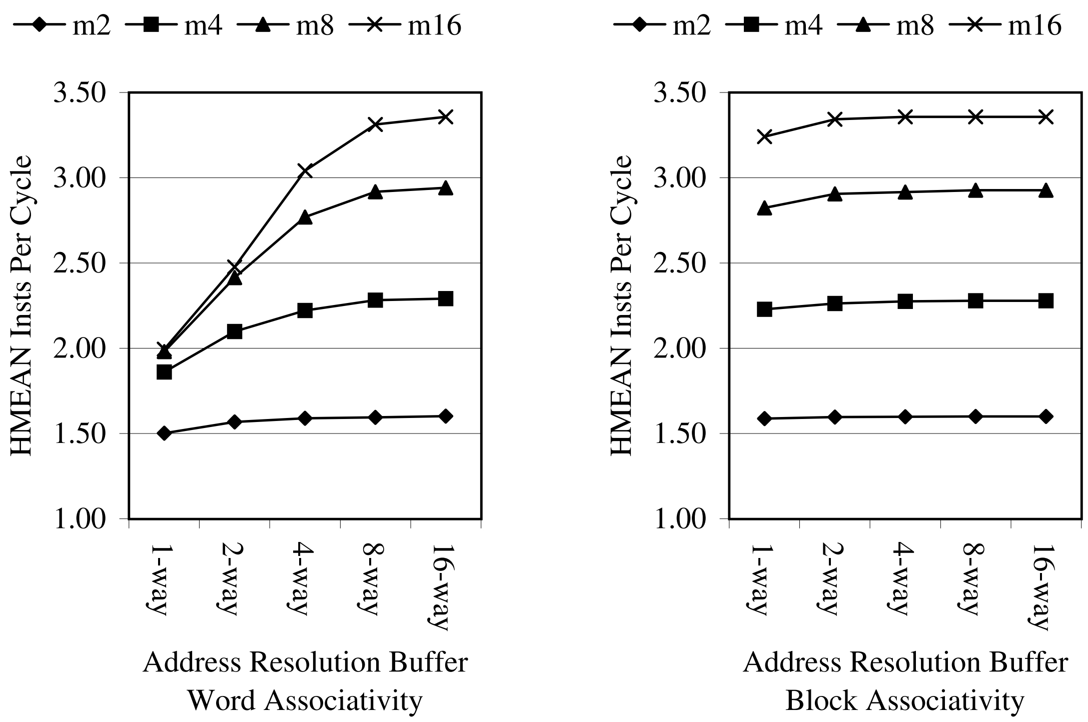


Figure 6.37: SPEC CINT95 performance of direct mapped to fully associative address resolution buffers with 16 word or block entries for 2, 4, 8, and 16 processing units.

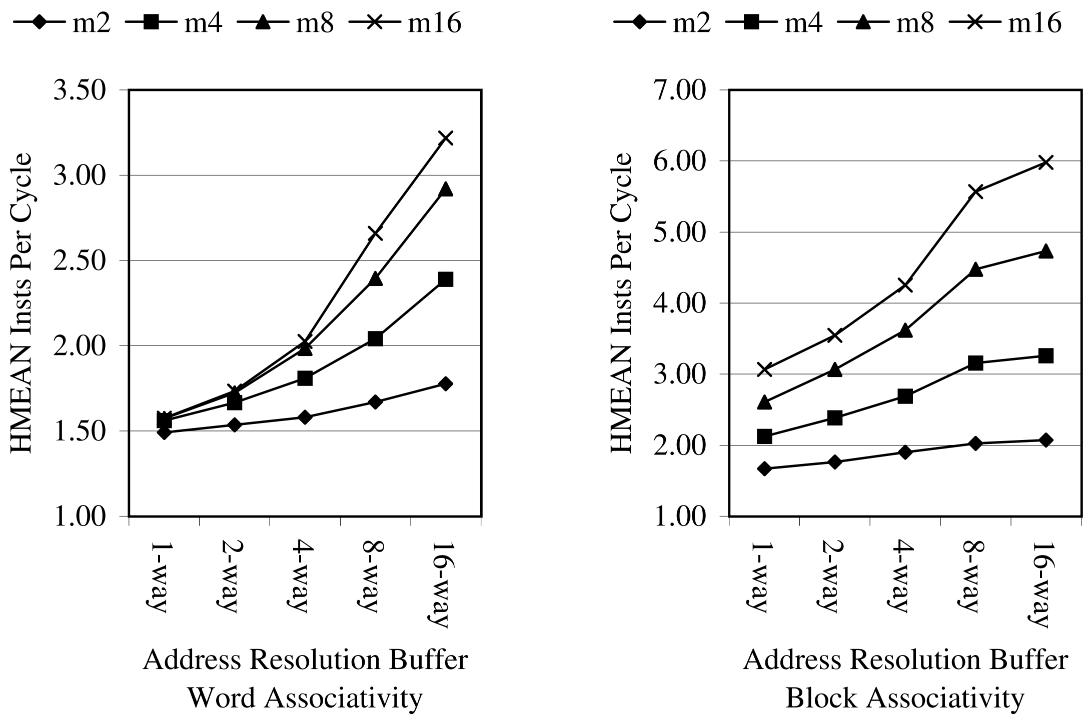


Figure 6.38: SPEC CFP95 performance of direct mapped to fully associative address resolution buffers with 16 word or block entries for 2, 4, 8, and 16 processing units.

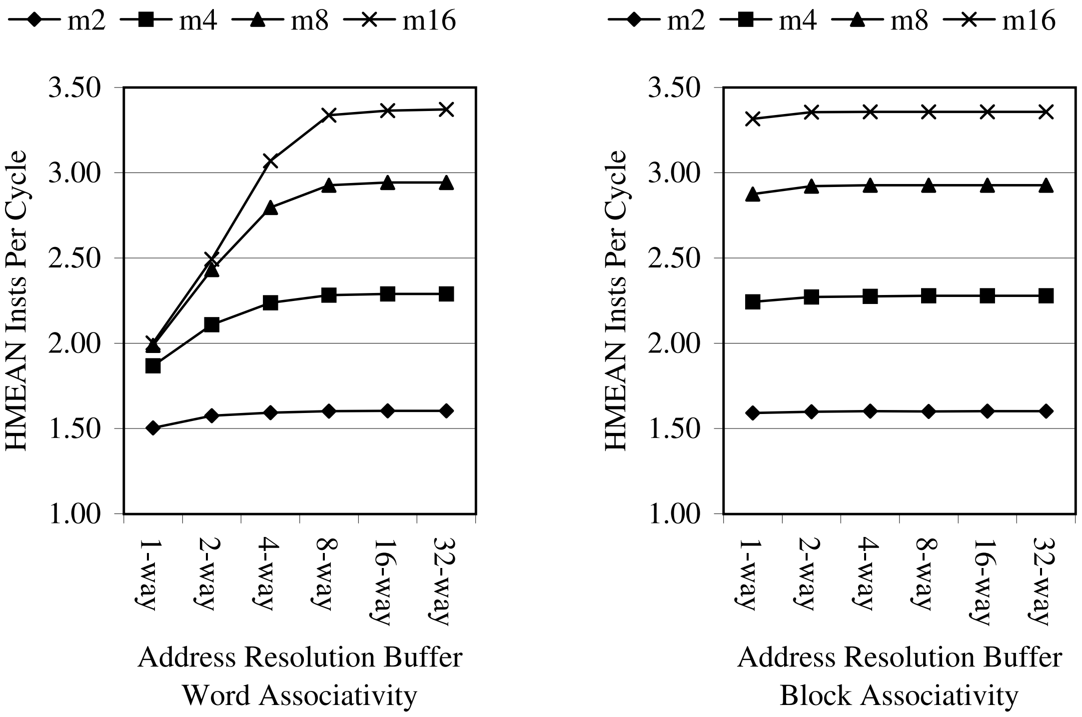


Figure 6.39: SPEC CINT95 performance of direct mapped to fully associative address resolution buffers with 32 word or block entries for 2, 4, 8, and 16 processing units.

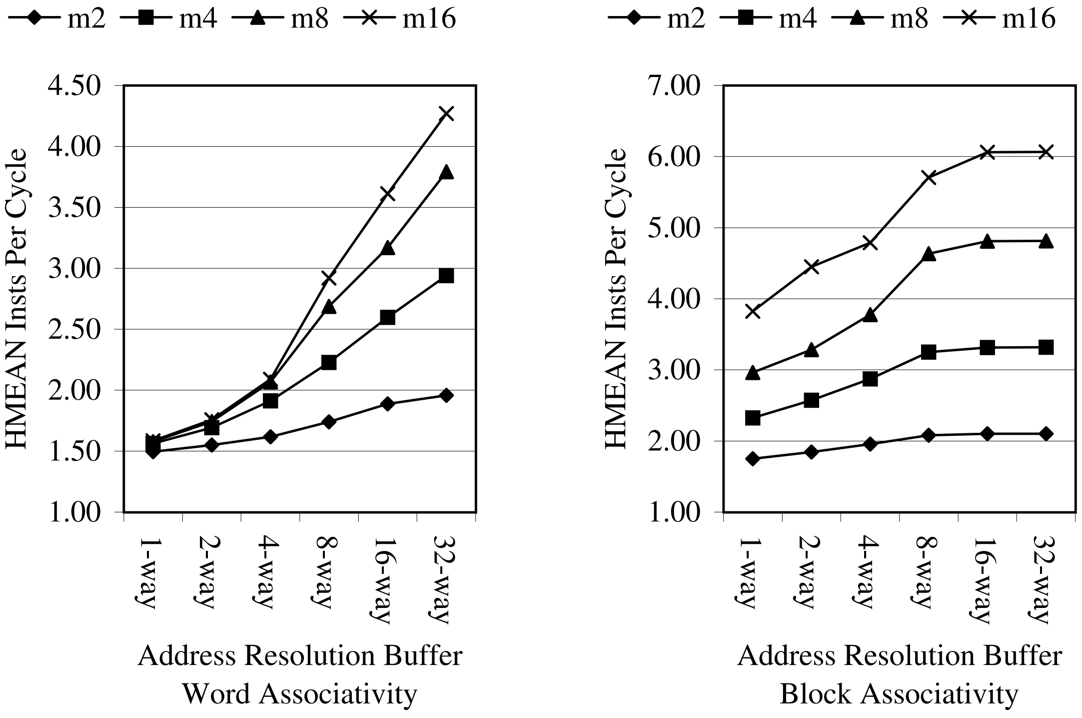


Figure 6.40: SPEC CFP95 performance of direct mapped to fully associative address resolution buffers with 32 word or block entries for 2, 4, 8, and 16 processing units.

more (by a factor of the number of words per cache, 8 in this case) of the memory namespace. Nonetheless, the difference in performance for low and high associativity, though still quite significant, is not as pronounced.

To focus just on relationship between the number of address resolution buffer entries (either word or block versions) and performance, a fully associative address resolution buffer may be considered. Figure 6.41 gives the performance of fully associative address resolution buffers with 8, 16, and 32 word (on the left) or block (on the right) entries for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations. Just below, Figure 6.42 gives the results for the SPEC CFP95 benchmarks with same configurations. For the integer benchmarks, the difference between the smallest (8 entries) and the largest (32 entries) word entry address resolution buffers is modest, though it does become more significant as number of processing units (and their demand for entries) increases. Furthermore, block entry address resolution buffers show no appreciable difference in performance between the smallest (8 entries) and the largest (32 entries) address resolution buffers. For the floating point benchmarks, the difference between the smallest (8 entries) and the largest (32 entries) word entry address resolution buffers is enormous, especially as the number of processing units (and their demand for entries) increases. (The performance almost doubles for most aggressive processor configurations.) In addition, it should be noted that the performance in this case does not level off, indicating that even the largest configuration studied is inadequate. In contrast, performance does eventually level off for the block entry address resolution buffers. However, there is still a significant performance difference between the smallest (8 entries) and the largest (32 entries) address resolution buffer, more for larger numbers of processing units and less for smaller numbers of processing units.

6.8.3.3 Memory Dependence Table Results

As a reminder, for the memory dependence table results, the size is fixed at 16 entries per bank with full associativity for runs when it is used and at 0 entries when it is not used, for all numbers of processing units.

Figure 6.43 gives the performance with and without a memory dependence table for the SPEC CINT95 benchmarks using 2, 4, 8, and 16 processing unit configurations (on the left) and running 2 and 4 tasks per processing unit, one-at-a-time and all-at-once, using the 4 processing unit configuration (on the right). Just below, Figure 6.44 gives the same information for the SPEC CFP95 benchmarks. Clearly, for both the integer and floating point benchmarks the dynamic synchronization of producers and consumers provided by the memory dependence table plays a significant role in terms of overall performance. Moreover, this role increases in significance as the size of the task window on the dynamic instruction sequence increases.

For the integer benchmarks, a configuration of 2 processing units (2 tasks) shows no appreciable difference in performance with or without the memory dependence table; yet, a configuration of 16 processing units (16 tasks) shows a significant difference in performance of nearly 1.00 instructions per cycle (40%) with and without the memory dependence table. Likewise, for the floating point benchmarks, a configuration of 2 processing units shows not performance difference, but a configuration of 16 processing units shows a difference

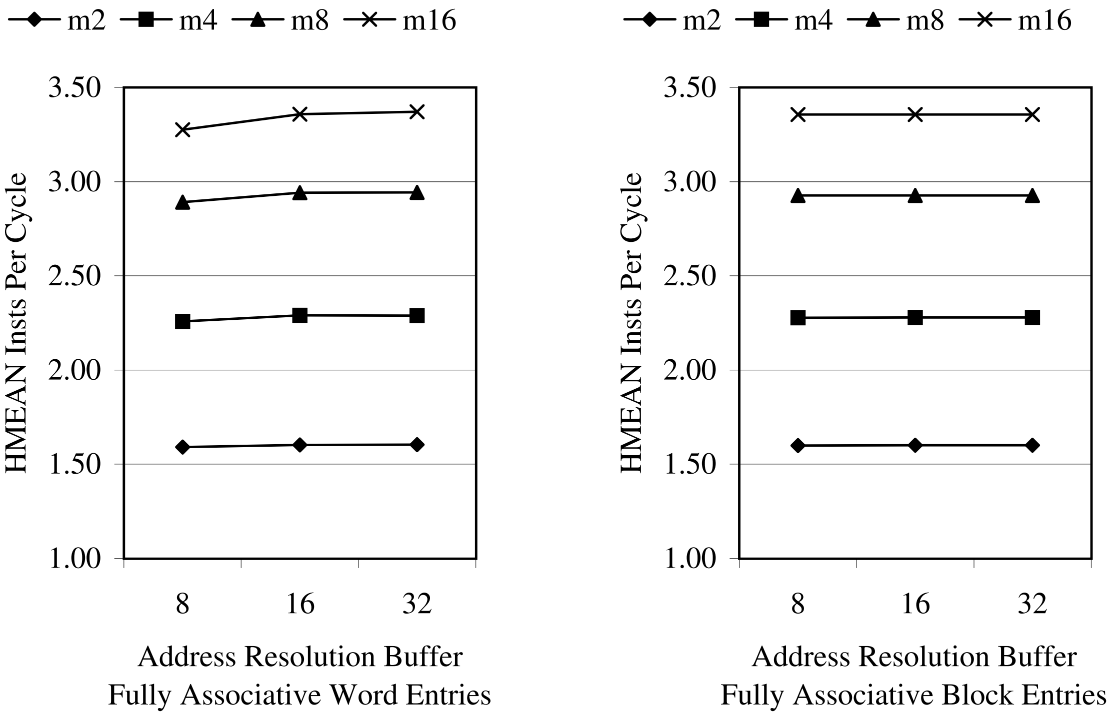


Figure 6.41: SPEC CINT95 performance of fully associative address resolution buffers with different numbers of word or block entries for 2, 4, 8, and 16 processing units.

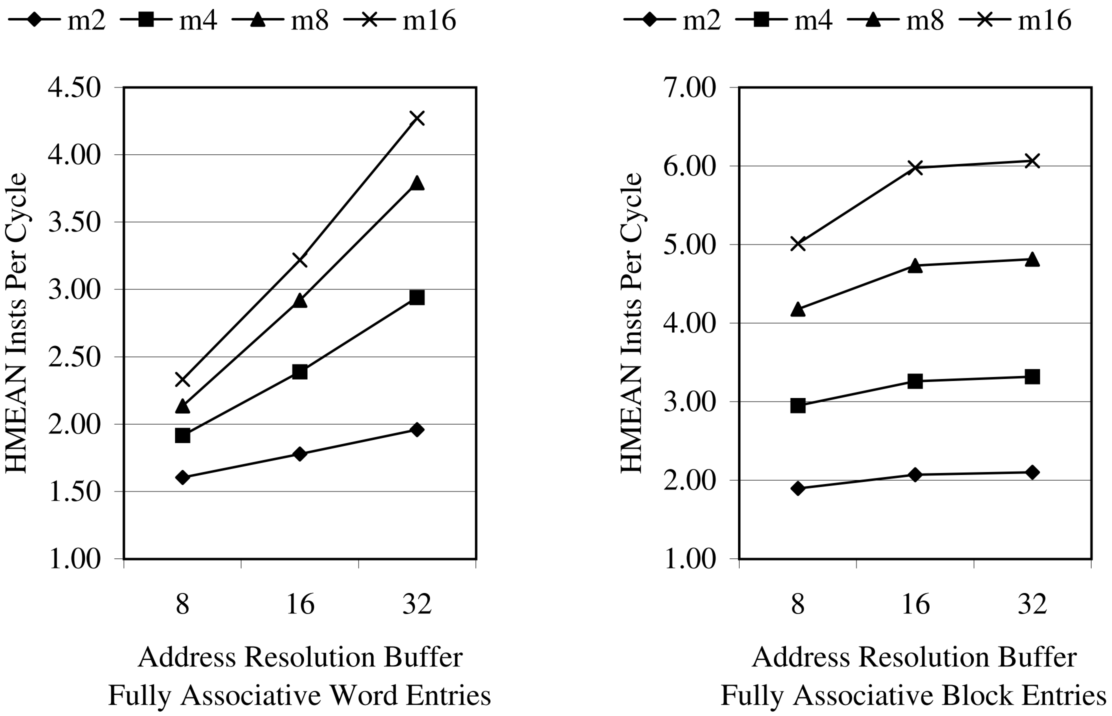


Figure 6.42: SPEC CFP95 performance of fully associative address resolution buffers with different numbers of word or block entries for 2, 4, 8, and 16 processing units.

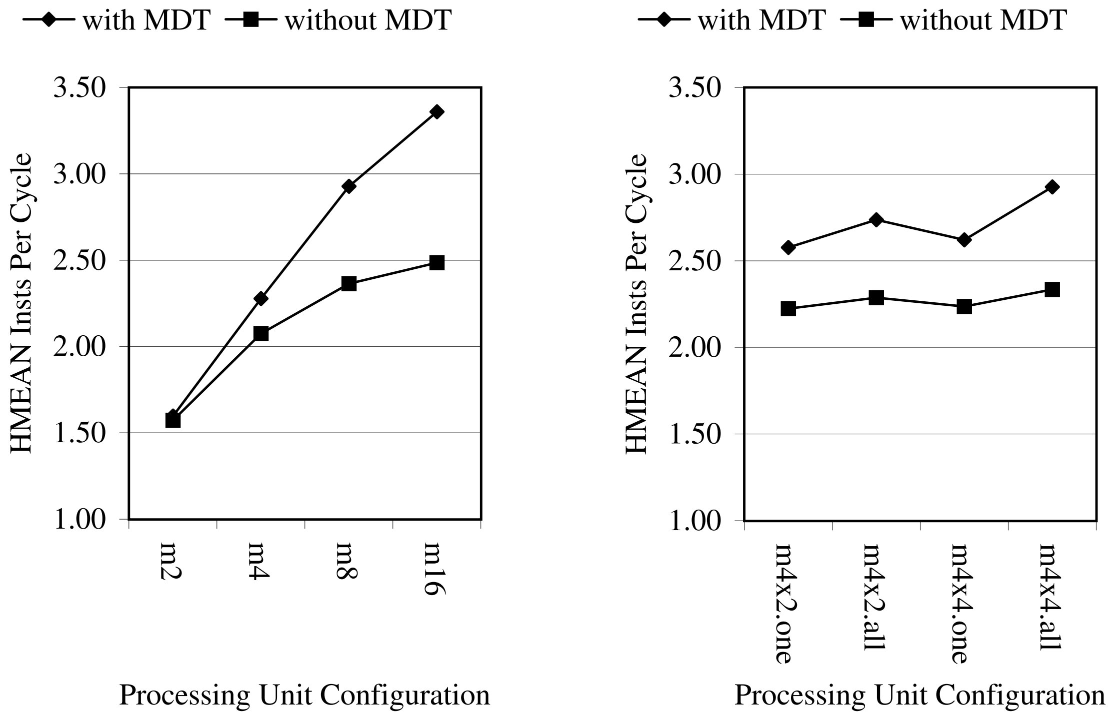


Figure 6.43: SPEC CINT95 performance without/without a memory dependence table for different processing unit configurations.

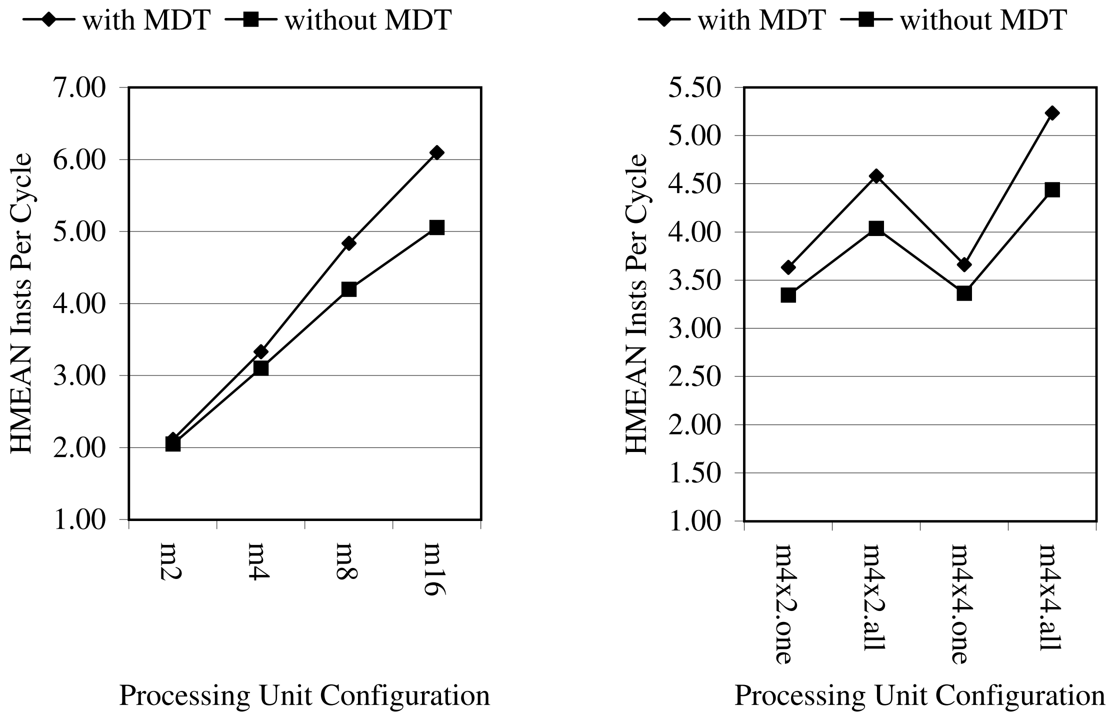


Figure 6.44: SPEC CFP95 performance without/without a memory dependence table for different processing unit configurations.

of a little over 1.00 instructions per cycle (20%) with and without the dependence table. Moreover, similar performance may be observed running multiple tasks per processing unit, with the effect of the memory dependence table more significant for all-at-once than for one-at-a-time.

6.9 Summary

This chapter studied the data supply of a multiscalar processor. It described how the difference between the register and memory namespaces inspired different solutions to the same data supply problem. That is, the architectural storage must appear as single logical (register and memory) namespaces to running programs, regardless of the actual physical design in order to support the serial program model. The key issues for a successful design of the data supply were identified as effective and efficient mechanisms to handle storage, communication, speculation, and synchronization. To this end, two distinct approaches, tailored for the characteristics of the respective namespaces, were taken to support the register file and data memory aspects of the data supply.

6.9.1 Register File

The register file was designed to rely on a combination of software and hardware to provide an organization that functioned effectively and efficiently, by placing the complexity in software rather than hardware. The software, *i.e.* the compiler, assisted the hardware by providing information, about the nature of register communication among tasks, that would otherwise not be available (delaying register file actions) or otherwise have to be computed at run-time (complicating register file design). The hardware consisted of a collection of relatively simple register files, duplicated among the processing units and connected via a point-to-point unidirectional communication ring. The evaluation of this design investigated the behavior in terms of various register traffic metrics as well as the overall performance for different register file latency and bandwidth characteristics.

6.9.1.1 Behavior

The behavior results were divided according to the particular aspect of register traffic under consideration. The register traffic was measured as the observed register communication from each register file to the next adjacent register file (on the links of the communication ring) summed among all register files. The sum included the register instances generated by a task and all previous tasks that must be communicated. The three aspects studied were (1) the register traffic per useful task and the register traffic per useful instruction, (2) the total register traffic and the link register traffic, and (3) the register task travel distance.

The register traffic per useful task and the register traffic per useful instruction were calculated as the register traffic divided by the number of useful tasks or useful instructions. The total register traffic per cycle was calculated as the register traffic over all communication links divided by the number of program execution cycles. The link register traffic per cycle was calculated as the total register traffic per cycle divided by the number of communication links. The register task travel distance was measured as the distance that a register instance traveled from the task that generated it, counted in terms of tasks from the dynamic instruction sequence.

Considering the register traffic per useful task and the register traffic per useful instruction for the SPEC CINT95 and SPEC CFP95 benchmarks, the register traffic increased steadily as the number of tasks in concurrent execution for both metrics. Moreover, the amount of register traffic per useful task and per useful instruction exceeded the traffic due solely to the register instances generated, with the amount of register traffic increasing as the number of processing units. Nevertheless, the amount of register traffic was held in check as seen by the fact that the register traffic was not doubled as the number of processing units was doubled. This behavior was attributed to that fact that many of the register instances generated by earlier tasks were filtered by later tasks as other register instances of the same register name were generated. In addition, the increases in register traffic due to increases in the number of processing units were observed to be more significant with mis-speculation than without mis-speculation. This behavior was attributed to the fact that mis-speculation occurred more often with more processing units, so more register traffic was generated as result.

Considering the total register traffic per cycle for the SPEC CINT95 and SPEC CFP95 benchmarks, it was observed that the total register traffic per cycle increased steadily as the number of tasks in concurrent execution increased, with mis-speculation as well as without mis-speculation. For processors that were configured to run only a few tasks, the total register traffic per cycle with mis-speculation and without mis-speculation was about the same since control and data dependence violations were less likely for fewer processing units. For processors that were configured to run many tasks, either one per processing unit or many per processing unit, the total register traffic per cycle was greater with mis-speculation than without mis-speculation, but not even by as much as 1.0 registers per cycle in the worst case.

Considering the link register traffic per cycle for the SPEC CINT95 and SPEC CFP95 benchmarks, it was observed that the link register traffic per cycle actually decreased as the number of tasks increased. This behavior was attributed to that fact that the number of processing units, and more significantly the number of communication links, increased as did the number of tasks. Yet, because the total register traffic did not increase at the same rate as the number links, the link register traffic decreased. In contrast, for multiple tasks per processing unit configurations the link register traffic per cycle indeed increased as the number of tasks increased, but only slightly. This behavior was attributed to the fact that the number of processing units, and likewise the number of communication links, did not change as did the number of tasks. Since the total register traffic increased, the link register traffic increased as well.

Comparing the register task travel distance for the SPEC CINT95 and SPEC CFP95 benchmarks, registers traveled farther as the number of tasks in concurrent execution increased, more so with mis-speculation than without, but registers still traveled only a few "hops" (between 1 and 4) from the tasks that generated them. The distance in register link "hops" that a register traveled might be as much as the number of tasks in concurrent execution (a full cycle of the tasks). Yet the results indicated that the register task travel distance was held in check as the number of tasks in concurrent execution grew, even taking into account the effects of mis-speculation.

Contrasting the register task travel distance for the SPEC CINT95 and SPEC CFP95 benchmarks, registers traveled almost twice as far for the integer benchmarks as for the floating point benchmarks. This behavior was due in part to the fact that integer and floating point tasks are much different in sizes and in part due to the fact that the communication patterns of the integer and floating point benchmarks are intrinsically different. In addition, the difference between the distances with mis-speculation and without mis-speculation was more significant for integer than for floating point benchmarks, but still less than 1 "hop".

6.9.1.2 Performance

Overall, the results indicated that the register file latency rather than the register file bandwidth was the critical factor in terms of the register file impact on processor performance. In particular, the register file latency had the effect of compressing the performance of the more aggressive processing unit configurations to that of the performance of the less aggressive processing unit configurations, more so for integer than for floating point benchmarks. There was a similar effect due to the register file bandwidth, but it was relatively insignificant in this study. That is, the register file bandwidth did have an effect on processor performance, though it was much less significant relative to the register file latency. To be specific, the processor performance improved steadily and at a fairly high rate as the register file latency was decreased; yet, the processor performance only improved marginally as the register bandwidth was increased and at a very small rate that leveled off rather quickly.

6.9.2 Data Memory

The data memory was designed to rely solely on hardware to provide an organization that functioned effectively and efficiently. A more or less conventional data cache, though aggressive in terms of its overall bandwidth and latency characteristics, was joined with two specialized components, the address resolution buffer and the memory dependence table, to deliver a design capable of supporting multiscalar execution.

This data memory was designed not only to provide high-bandwidth and low-latency operation (via the data cache), but to allow accesses to occur in a speculative manner without knowledge of preceding loads or stores (via the address resolution buffer). Moreover, it was designed to gather the dependence relationships between speculated loads and stores dynamically in order to synchronize those dependent loads and stores (via the memory dependence table) whose execution would otherwise lead to dependence violations and performance losses .

Overall, the characteristics of the data caches, the conventional components, determined the level of performance that the data memory was capable of providing, while the characteristics of the address resolution buffer and the memory dependence table, the specialized components, determined to what extent the data memory was able to achieve it. Moreover, the effects of these specialized components were isolated into those attributable to the address resolution buffer and those attributable to the memory dependence table.

6.9.2.1 Data Cache

Considering the data cache experiments, it was found that the data cache exhibited the same kind of typical cache-like behavior that the task and instruction caches did with respect to the factors of (1) miss rate, (2) associativity, and (3) size. First, the miss rate decreased steadily as the data cache size increased and leveled off once the working set of the program data fit into the data cache. Second, the use of associativity improved the miss rate of the data cache, but this improvement diminished as the associativity was changed from 1-way associativity to 2-way associativity and then from 2-way to 4-way associativity. Third, as the data cache size was doubled for both the SPEC CINT95 and SPEC CFP95 benchmarks, the performance in terms of instructions per cycle increased moderately but steadily.

Despite the common behavior attributed to these factors among the different types of caches for both the floating point and integer benchmarks, the effective load latency behavior of the data cache distinguished it from the other types of caches studied in this work. The effective load latency remained relatively constant across processing unit configurations for the SPEC CINT95 benchmarks. However, the effective load latency increased by as much as a cycle each time the number of processing units was doubled for the SPEC CFP95 benchmarks. This behavior was attributed to the fact that the higher levels of parallelism in the floating point benchmarks compared to integer benchmarks resulted in higher demand for data memory bandwidth.

This higher demand in turn caused higher contention at the cache banks and subsequently a higher number of cache bank conflicts which could not be accommodated by cache block combining. Nevertheless, such higher levels of parallelism afforded the floating point benchmarks greater latency tolerance than the integer benchmarks, as evidenced by the fact that the performance declined more rapidly for the integer benchmarks (about 12% per cycle) than for the floating point benchmarks (about 3% per cycle) when the load hit latency increased. Neither the task nor instruction caches exhibited such behavior as the number of processing units was increased.

Yet, overall, the experiments indicated that a reasonable sized data cache (64K-byte) with a moderate amount of associativity (2-way) could provide nearly the highest performance possible on the SPEC CINT95 and SPEC CFP95 for the processor configurations under consideration. Nonetheless, in order to determine whether further improvements in instruction cache operation (aside from changing the hit latency) would provide further improvements in overall processor performance, an ideal data cache was compared with the real shared data caches.

The ideal data cache provided a significant performance improvement for floating point benchmarks, but only a modest performance improvement for integer benchmarks. In both cases, though, the improvement was due to the infinite bandwidth (which benefited the floating point far more than the integer benchmarks) rather than the infinite storage aspect of the ideal data cache (as miss rates were low and conflicts were high, relatively speaking), indicating that there is need for further consideration of the bandwidth and latency aspects of data cache designs (with respect to the benchmarks studied).

6.9.2.2 Address Resolution Buffer

Considering the address resolution buffer experiments, the critical performance factor was identified to be whether or not overflows were avoided, because the result of an overflow was a performance loss due to processor stall cycles. As the address resolution buffer tracks the loads and stores that occur speculatively, it needs storage to record this information. The overflow condition occurs when all of the address resolution buffer entries in a set are in use (have load and/store bits). Because an entry cannot be claimed until after the task associated with the loads and stores has retired, subsequent accesses are stalled. Thus, the address resolution buffer results focused on the impact of different address resolution buffer sizes and associativities with various processing unit configurations. The address resolution buffer results also focused on the impact of the amount of per entry version storage to illustrate the effect that this characteristic can have on performance in conjunction with the number of entries and their associativity.

For the SPEC CINT95 benchmarks, word entry address resolution buffers provided similar increases in performance for all processing unit configurations as the associativity increased. Moreover, the rate for the performance increase was directly correlated to the aggressiveness in number of processing units of the processor configuration. The reason for this behavior was that a smaller configuration placed less demand on the address resolution buffer than a larger configuration did; so, the larger configuration benefited more from the flexibility of higher associativity (in terms of stalls) than the smaller configuration. However, there seemed to be a knee above 8-way associativity where there was no apparent benefit in further associativity increases. In contrast to the word entry results, the block entry results showed only modest if any effect from increases in associativity. In addition, this effect diminished (to the point of being practically non-existent) as the number of block entries was increased.

For the SPEC CFP95 benchmarks the performance was more sensitive to the address resolution buffer than for the SPEC CINT95 benchmarks, using word as well as block entries. With respect to word entry configurations, the performance always appeared to be suppressed (never leveling off, and implying the need for more entries) for all of number of entries studied. Though the performance was suppressed for all sizes, it was particularly so for associativities below 4-way; for associativities of 4-way and above, the rate of performance improvement seemed to accelerate rapidly. It was concluded that the word entry design simply could not provide enough storage, at any associativity, to satisfy the needs of the floating point benchmarks. However, this effect was not limited to the word entry address resolution buffers. It also occurred for the block entry address resolution buffers, which could support quite a bit more (by a factor of the number of words per cache block, 8 in this case) of the memory namespace. Nonetheless, the difference in performance for low and high associativity with block entries, though still quite significant, was not as pronounced as with word entries.

In order to focus just on the relationship between the number of address resolution buffer entries (either word or block versions) and performance, a fully associative address resolution buffer was studied. For the SPEC CINT95 benchmarks, the difference between the smallest (8 entries) and the largest (32 entries) word entry address resolution buffers was modest, though it did become more significant as number of processing units (and their demand for entries) increased. The block entry address resolution buffers showed no appreciable difference in performance between the smallest and the largest address resolution buffers. For the SPEC CFP95 benchmarks, the difference between the smallest (8 entries) and the largest (32 entries) word entry address resolution buffers was enormous, especially as the number of processing units (and their demand for entries) increased. In fact, the performance did not level off, indicating that even the largest configuration studied was inadequate. In contrast, performance did eventually level off for the block entry address resolution buffers. However, there was still a significant performance difference between the smallest and the largest address resolution buffers.

6.9.2.3 Memory Dependence Table

Considering the memory dependence table experiments, the critical performance factor is whether or not loads and stores are synchronized, because the result of load/store dependence violations is performance loss due to processor squash cycles. Though the address resolution buffer tracks the loads and stores that occur speculatively, it does not attempt to ensure that the loads and stores occur according to program order. Instead, it detects when loads and stores occur out of program order and corrects these dependence violations by discarding all subsequent work (useful or non-useful). The memory dependence table tracks the dependences between those loads and stores that have caused dependence violations in the past in order to synchronize their future executions. The purpose of this synchronization is to prevent dependence violations and thereby avoid any resulting performance loss. The experiments performed on the memory dependence table focused on the performance impact with or without its use, instead of the specifics of its organization.

Both the SPEC CINT95 and SPEC CFP95 benchmarks showed that the dynamic synchronization of producers and consumers provided by the memory dependence table played a significant role in terms of overall performance. Moreover, this role increased in significance as the size of the task window on the dynamic instruction sequence increased. For the SPEC CINT95 benchmarks, a configuration of 2 processing units (2 tasks) showed no appreciable difference in performance with or without the memory dependence table; yet, a configuration of 16 processing units (16 tasks) showed a significant performance difference of nearly 1.00 instructions per cycle or 40% with and without the memory dependence table. Likewise, for the SPEC CFP95 benchmarks, a configuration of 2 processing units showed no performance difference, but a configuration of 16 processing units showed a difference of a little over 1.00 instructions per cycle or 20% with and without the memory dependence table. Moreover, similar performance differences were observed running multiple tasks per processing unit, with the effect of the memory dependence table more significant for all-at-once than for one-at-a-time.

Chapter 7

Performance Comparison

This chapter provides a performance comparison between realistic multiscalar processors and idealistic superscalar processors. Section 7.1 gives an overview of the performance comparison that is the subject of this chapter. Section 7.2 reflects on those performance factors taken into account and those not taken into account by the experimental framework used for this thesis. Section 7.3 provides the actual comparison between the performance of realistic multiscalar processors and idealistic superscalar processors. Section 7.4 puts all of the results from this performance comparison in perspective by considering the scenarios in which multiscalar processors might offer advantages over superscalar processors. Section 7.5 gives a summary of this chapter.

7.1 Overview

It has been the contention throughout this work that multiscalar processors are well-suited in terms of their engineering and performance characteristics to serve as the underlying technology of future microprocessors. Given this assumption, this work has focused on designing the components of multiscalar processors and on evaluating the performance levels of which multiscalar processors are capable for different configurations of those components. (This design and evaluation was the subject of the previous three chapters.)

7.1.1 Basics

Though the design and evaluation described thus far in this thesis is useful to ascertain how different multiscalar processors behave, it does not provide any perspective on how the behavior of multiscalar processors differs from other processors. In particular, it does not address the most important aspect in terms of comparing the behavior of different processors, namely performance. This chapter provides a performance comparison between realistic multiscalar processors and idealistic superscalar processors to gauge (at least to some extent) how the alternative multiscalar approach fairs against the conventional superscalar approach.

It is important to establish that this performance comparison is not intended to answer questions concerning the superiority of one processing paradigm over another – in this case the multiscalar paradigm over the superscalar paradigm or vice versa – as this issue is beyond the scope of this work. Instead, its sole intent is to provide performance expectations for realistic multiscalar processors in terms that have meaning to those interested in future microprocessor designs. Of particular concern, though, is the choice of a meaningful standard upon which to base a comparison. For the purpose of this study, a superscalar processor provides a useful standard because its operation is relatively well understood and documented in the literature.

Though superscalar processors are the present state of the art in processor technology, existing designs do not possess capabilities for extracting instruction-level parallelism (in terms of issue-width, instruction window size, and clock frequency) that are comparable to those expected for multiscalar processors like those studied in this thesis. Nevertheless, for the performance comparison to provide insight for future designs, it must be based on processor designs that have (at least somewhat) equivalent instruction-level parallelism potential. As a result, the superscalar processors used for the comparison have been given capabilities beyond the scope of existing designs.

In order to provide these capabilities in a straightforward and flexible manner, yet retain the characteristics of superscalar operation, it is necessary to do so using an idealistic version of a superscalar processor. This idealistic superscalar processor provides the desired operational characteristics, but does so without addressing how to (or whether it is even practical to) design a wide-issue, out-of-order, superscalar processor that supports a large instruction window and a high clock frequency. Because the goal of this work is to offer an alternative to conventional approaches, such as superscalar processors, its intention is not take on the problem of solving their obvious shortcomings. Instead, its intention is to provide the highest possible level of superscalar performance – one which no realistic superscalar processor is ever likely to achieve – against which to measure multiscalar performance.

7.1.2 Insights

The results in this chapter provide a performance comparison, relative to a baseline 1-wide out-of-order issue processor, between realistic multiscalar processors and idealistic superscalar processors (to the extent possible given the experimental framework used in this thesis). The overall results of this study indicate that realistic multiscalar processors are able to outperform idealistic superscalar processors over nearly the entire range of configurations for the SPEC CFP95 programs, but over only a selected range of configurations for the SPEC CINT95 programs (given the software and hardware assumptions upon which it is based). Furthermore, the coverage of these ranges is dependent upon the multiscalar processor configuration.

In particular, the coverage of these ranges increases as the number of processing units increases when running a single task per processing unit (one-at-a-time), and similarly as the number of tasks per processing unit increases when running multiple tasks per processing unit (all-at-once). Moreover, considering the cases where performance advantages exist, the ability of the multiscalar processor to provide better performance than the superscalar processor is influenced first by the number of tasks in concurrent execution on the processor and second by the overall issue width of the processor, though one without the other is not sufficient. The basis of this claim are two key observations from this performance comparison.

First, the multiscalar configurations with the highest performance across both the SPEC

CINT95 and SPEC CFP95 benchmarks are those with the highest number of tasks in concurrent execution. Consistent with this observation, the performance decreases as the number of tasks in concurrent execution decreases, while the performance increases for the same number of tasks as the number of processing units increases. Second, with respect to any particular processor configuration, higher performance is attained with higher processor (or actually processing unit) issue width; however, processor configurations with more overall issue width but less tasks in concurrent execution are outperformed in general by processor configurations with less overall issue width but more tasks in concurrent execution.

7.2 Performance

With the overview of the performance comparison just presented in mind, it is worth switching gears to reflect upon what aspects of performance the comparison provided in this chapter actually takes into account. The classic definition of performance in terms of the time to execute a program is given by the following equation:

$$\frac{time}{prog} = \frac{inst}{prog} * \frac{cycle}{inst} * \frac{time}{cycle}$$

This equation identifies three major factors contributing to performance, the time per program: (1) the instructions per program, (2) the cycles per instruction, and (3) the time per cycle. The instructions per program is determined by the compiler as well as the algorithms contained in the original program. The cycles per instruction is the reciprocal of the instructions per cycle, the metric used throughout this work for the amount of instruction-level parallelism extracted by a processor. Because of the reciprocal relationship, the time per program decreases as the instruction-level parallelism given as instructions per cycle increases. The time per cycle is the reciprocal of the clock frequency, or the clock period, of a processor. Again, because of the reciprocal relationship, the time per program decreases as the clock frequency increases.

Though all of these factors can be accounted for in an actual implementation, a compiler and processor, the experimental framework used for this thesis is based on a compiler and simulator. These tools are only designed to provide accurate instruction and cycle counts of program execution on different processor models. As described in Chapter 3, these models do not take into account the aspect of clock frequency in the performance equation. There is no question that this aspect is an important one. However it is simply beyond the scope of this work to account for it because the level of processor design detail required is too high. To elaborate, it is usually difficult to provide an accurate estimate of clock frequency without a full processor implementation in a specific technology (obviously, more than is the intent of this work).

Nevertheless, it is often acceptable to compare performance without factoring in clock frequency, if the designs being compared are similar with respect to their fundamental underpinnings (*e.g,* two different superscalar implementations). In these cases, it is assumed that the differences in the two alternatives are related solely to the instructions per program from the compiler or the instructions per cycle from the processor; the clock frequency of the two implementations is assumed to be the same. However, the key factors which form the basis for the expected performance advantages of multiscalar processors are higher levels of instruction-level parallelism and higher clock frequencies in actual implementations. Furthermore, the expected performance advantages of the multiscalar approach are as likely to come from both of these factors collectively as well as from one or the other individually.

As a result, the performance comparison presented here may be misleading. In cases where the primary performance advantage is due to clock frequency the results presented in this chapter might not accurately portray the differences between multiscalar processors and superscalar processors. Moreover, this discrepancy is of particular concern because while the multiscalar processors used for this comparison are realistic, the superscalar processors are idealistic. Since there is no perfect solution to this problem, the results of this performance comparison must stand on their own (as most often do), with the obvious caveat that "your mileage may vary". Even so, the section of this chapter following the performance comparison endeavors to account for the impact of clock frequency by using the results to describe the scenarios in which clock frequency plays a key role in whether multiscalar processors can provide performance advantages over superscalar processors.

7.3 Comparison

The purpose of this comparison is to provide insight as to where the realistic multiscalar processors studied in this thesis stand relative to idealistic superscalar processors in terms of performance. Nevertheless, it must be emphasized that the data provided here for multiscalar processors is not based on an actual implementation but instead on a simulation model; so, the "true" behavior may be different (either more or less). Likewise, the data provided for the superscalar processor is also based on a simulation model rather than actual implementation; so, there are similar caveats about the "true" performance. (A brief distinction between realistic and idealistic is made in the next paragraph; a more comprehensive distinction follows in a later section.)

The multiscalar processors used for this comparison are considered "realistic" because their designs take into account the forecasted characteristics of future technology. That is, communication and computation are handled in a manner that can foster an actual implementation which uses large instruction windows, wide instruction issue, and high clock frequencies. In contrast, the superscalar processors used for this comparison are considered "idealistic" because their designs, though (more or less) comparable in capability to the multiscalar designs in terms of instruction window size and instruction issue width, do not accommodate future technology characteristics. Therefore, these designs do not handle communication and computation in a manner that is suitable for an actual implementation.

7.3.1 Metrics

As has been emphasized throughout this thesis, the fundamental metric with which to evaluate processor performance is time. Yet, because the details that impact the actual cycle time of a processor are not modeled by the simulator used for these studies, performance has thus far been measured in terms of instruction-level parallel execution given as the number of useful instructions completed per cycle (or just instructions per cycle). This metric is appropriate when the performance of different multiscalar processor configurations are studied because each uses the same program binary and executes the same number and sequence of instructions for a given input. However, this metric is not an appropriate one to compare the performance of multiscalar processors with superscalar processors because the program binaries used and hence the number and sequence of instructions executed are different.

As described in Chapter 3 and as detailed in Tables 3.3 and 3.4, a multiscalar processor executes an average of around 5% more instructions for the SPEC CINT95 programs and 2% more instructions for the SPEC CFP95 programs than a superscalar processor. With different instruction counts, the instructions per cycle is an incorrect metric with which to compare multiscalar and superscalar performance. Therefore, this performance comparison defines a baseline processor and measures performance as speedup over this baseline processor. The speedup is a ratio given by the number of cycles to execute a particular benchmark and its input with the baseline processor divided by the number of cycles to execute the same benchmark and its input with the superscalar or multiscalar processors, respectively.

This speedup is given as the unweighted harmonic mean (HMEAN) for each of the two groups of benchmarks, SPEC CINT95 and SPEC CFP95, respectively. It is important important to point out that though the benchmark binaries for the superscalar and multiscalar processors are different, the actual benchmarks and the inputs for each are the same. (The baseline scalar processor uses the same binary as the superscalar processor, but is restricted to issue only one instruction each cycle.) As with instructions per cycle, the harmonic mean (HMEAN) is the correct mean (as opposed to the arithmetic or geometric) because the speedup is inversely proportional to program execution time, and the speedup should not be weighted because each benchmark program is meant to be representative of a particular type of (integer or floating point) application independent of how long it executes for its particular input.

7.3.2 Configurations

As discussed above, the performance comparison described in this chapter involves three different processor configurations: the baseline processor configuration, the superscalar processor configuration, and the multiscalar processor configuration. In addition, there are two different sets of benchmark binaries: the superscalar binaries and the multiscalar binaries. The superscalar binaries are used by the baseline and superscalar configurations, while the multiscalar binaries are used only by the multiscalar configuration.

7.3.2.1 Baseline

The instruction and data processing of the baseline processor is configured as 1-wide, out-of-order issue supported by a 64 entry in-flight queue and a 16 entry on-deck queue. The functional unit latencies are exactly the same as those described in Chapter 3 and detailed in Table 3.12. The baseline processor is designed to provide performance of around 1 instruction per cycle. Table 7.1 gives the performance in instructions per cycle of the baseline processor on both the SPEC CINT95 and SPEC CFP95 benchmarks. The harmonic mean among all of the integer or floating point programs is given at the bottom of the table.

The instruction supply for the baseline processor consists of a predictor and an instruction cache.

The global-pattern-based predictor predicts among 2 targets and uses a pattern register that contains N=16 target patterns with B=1 bits from each control flow point for a total of 16 bits. The pattern register is XORed with the low order 16 bits of the supplied address from the control flow point being predicted. This configuration produces 16 bits to index the 64K-entry prediction state of 2-bit counters as well as other information used by the predictor.

The instruction cache is configured as 64K-byte with 32 byte blocks using 2-way associativity and least recently used replacement. Each access to the instruction cache is for 4 contiguous instructions for the address given. The access latency for the instruction cache is 1 cycle on a hit. A miss to the next level unified cache is an additional 10 cycles without contention; a further miss to the main memory is an additional 50 cycles without contention. The instruction cache is a lockup-free cache that supports unrestricted hits and supports 1 primary misses per bank as well as 3 secondary misses per primary miss.

The data supply for the baseline processor consists of a data cache with an ideal load/store queue of unlimited size for disambiguation. The data cache is configured as 64K-byte with 32 byte blocks using 2-way associativity and least recently used replacement. Each access to the data cache is for a word (or some part of a word) for the address given. The access latency for the data cache is 2 cycles on a hit (pipelined where the pipeline is flushed on a miss). A miss to the next level unified cache is an additional 10 cycles without contention; a further miss to the main memory is an additional 50 cycles without contention. The data cache is a lockup-free cache that supports unrestricted hits and supports 8 primary misses per bank as well as 3 secondary misses per primary miss.

7.3.2.2 Superscalar

The instruction and data processing of the superscalar processor is configured in one of three ways: 1) as uniform 4-wide, out-of-order issue supported by a 64 entry in-flight queue and a 16 entry on-deck queue, 2) as uniform 8-wide, out-of-order issue supported by a 128 entry in-flight queue and a 32 entry on-deck queue, or 3) as uniform 16-wide, out-of-order issue supported by a 256 entry in-flight queue and a 64 entry on-deck queue. These configurations provide increasingly aggressive issue characteristics with corresponding increasingly aggressive instruction window characteristics. The functional unit latencies are the same as for the baseline processor.

The instruction supply for the superscalar processor consists of a predictor and an instruction cache.

The global-pattern-based predictor predicts among 2 targets and uses a pattern register that contains N=16 target patterns with B=1 bits from each control flow point for a total of 16 bits. The pattern register is XORed with the low order 16 bits of the supplied address from the control flow point being predicted. This configuration produces 16 bits to index the 64K-entry prediction state of 2-bit counters as well as other information used by the predictor.

SPEC CPU95 Integer Benchmark	Baseline Insts Per Cycle
099.go	0.89
124.m88ksim	0.97
126.gcc	0.91
129.compress	0.96
130.li	0.93
132.ijpeg	0.97
134.perl	0.97
147.vortex	0.98
	0.95

SPEC CPU95 Floating Point Benchmark	Baseline Insts Per Cycle
101.tomcatv	0.92
102.swim	0.95
103.su2cor	0.95
104.hydro2d	0.95
107.mgrid	0.97
110.applu	0.99
125.turb3d	0.97
141.apsi	0.96
145.fpppp	0.91
146.wave5	0.92
	0.95

Table 7.1: Baseline performance for the SPEC CINT95 and SPEC CFP95 benchmarks.

The instruction cache is configured as 64K-byte with 32 byte blocks using 2-way associativity and least recently used replacement. Each access to the instruction cache is for 4 contiguous instructions for the address given. The access latency for the instruction cache is 1 cycle on a hit. A miss to the next level unified cache is an additional 10 cycles without contention; a further miss to the main memory is an additional 50 cycles without contention. The instruction cache is a lockup-free cache that supports unrestricted hits and supports 1 primary misses per bank as well as 3 secondary misses per primary miss.

This configuration is the same as for the baseline processor except in the following ways. The 4-wide superscalar processor performs 1 predict-access sequence each cycle (just like the baseline), but the 8-wide superscalar processor performs 2 predict-access sequences each cycle (back-to-back), and the 16-wide superscalar processor performs 3 predict-access sequences each cycle (back-to-back). These configurations provide increasingly aggressive instruction supply characteristics (though in an idealistic manner) with corresponding increasingly aggressive processing characteristics.

The data supply for the superscalar processor consists of a data cache with an ideal load/store queue of unlimited size for disambiguation. The data cache is configured as 64K-byte with 32 byte blocks using 2-way associativity and least recently used replacement. The storage is divided evenly among 16 banks and is interleaved by cache block. The functional units are connected to the data caches via a crossbar, which allows combining of concurrent accesses to the same cache block. Each access to the data cache is for a word (or some part of a word) for the address given. The access latency for the data cache is 2 cycles on a hit. A miss to the next level unified cache is an additional 10 cycles without contention; a further miss to the main memory is an additional 50 cycles without contention. The data cache is a lockup-free cache that supports unrestricted hits and supports 8 primary misses per bank as well as 3 secondary misses per primary miss.

7.3.2.3 Multiscalar

The multiscalar processor used in this performance comparison is nearly the same as the standard configuration that has been used throughout this thesis. Briefly, the standard configuration provides significant, but not unlimited instruction and data processing in the form of uniform (any instruction type) 1-wide, 2-wide, or 4-wide out-of-order processing units supported by a 16 entry in-flight queue and an 8 entry on-deck queue. The instruction supply and the data supply are aggressive enough to support this processing capacity, but remain faithful to the behavior of actual implementations (no perfect bandwidth or latency characteristics are assumed). The register file propagation latency is 1 cycle for each “hop”, and the propagation bandwidth (in register values per cycle) is equal to the issue width (in instructions per cycle). The hit latency for the top level instruction and task caches is 1 cycle; the hit latency for the top level data cache is 2 cycles. The miss latency for all top level caches is 10 cycles without contention to the unified next level cache. The main memory access latency beyond the unified next level cache is 50 cycles without contention. The details of the exact configurations are given in Table 3.11 of Chapter 3.

Overall there are two basic multiscalar processor configurations considered in this performance comparison. The first configuration uses 4, 8, or 16 processing units that run a single task per processing unit. The second configuration uses 4 processing units that run multiple tasks per processing unit all-at-once (one-at-a-time is not considered here). Moreover, for both of the multiscalar processor configurations, this performance comparison considers two different data supply latency configurations; one assumes aggressive latency characteristics, and the other assumes conservative latency characteristics. The aggressive latency characteristics are the same as those that have been assumed throughout this thesis. Namely, the register propagation delay is 1 cycle for each “hop”, and the data cache hit latency is 2 cycles. The conservative latency characteristics change these assumptions. In particular, the register propagation delay is 2 cycles for the first “hop” with 1 cycle for each “hop” thereafter, and the data cache hit latency is 3 cycles. These different assumptions result in a performance difference range of roughly 10% to 15% for SPEC CINT95 and 0% to 5% for SPEC CFP95 depending on the processor configuration and the benchmark.

7.3.3 Realistic/Idealistic Distinction

The key difference between the realistic and the idealistic designs used in this comparison is that the performance of the realistic design (for the multiscalar processor) takes into account the overheads involved in accommodating the characteristics of the implementation technology, while the idealistic design (for the superscalar processor) does not. These overheads have the positive effect of making a design practical. However, these overheads also have the negative effect of degrading the performance of a design (relative to one that does not have these overheads). It is also important to keep in mind that the experimental framework is limited in its capacity to account for all possible beneficial as well as detrimental effects of compiler optimizations in both the multiscalar and superscalar cases.

With respect to this distinction between realistic and idealistic, care must be taken when interpreting the data from this study for the following two reasons. First, this study does not account for the performance differences – as a result of clock frequency and instruction-level parallelism differences – in an actual implementation that distinguishes a realistic design from an idealistic one. Second, this study represents only one hardware (processor organization) and software (compiler capability) design point in both the multiscalar and superscalar processor design spaces. Since this work focuses on the hardware rather than the software it is most appropriate to comment on the former rather than the later aspect of the performance comparison.

In this study, the multiscalar processors considered are deemed realistic because their design *does take into account* the expected constraints of future implementation technology. In particular, the multiscalar processors studied here rely heavily on (1) the duplication of small, regular structures and (2) the decentralization of critical resources to enhance their overall performance capabilities. This approach favors local computation and communication over global to accommodate the characteristics of future implementation technology where communication as much as computation may be the performance limiter.

On the other hand, the superscalar processors considered in this study are deemed idealistic because their design *does not take into account* the expected constraints of future implementation technology. The main reason for this discrepancy is that how to (or whether it is practical to) design a superscalar processor with the capabilities needed for comparison is still an open research question. As a result, the superscalar processors studied here rely on (1) the extension of large, complex structures and (2) the centralization of critical resources to enhance their overall performance capabilities. Because such a superscalar design is not likely to be implementable, it is important to keep that in mind when drawing conclusions with respect to the multiscalar designs as well as the superscalar designs.

7.3.4 Results

The results presented here are divided into two parts (based on the data supply latency characteristics described earlier in 7.3.2.3). The first part considers the results comparing a realistic multiscalar processor with *conservative* latency characteristics to an idealistic superscalar processor. The second part considers the results comparing a realistic multiscalar processor with *aggressive* latency characteristics to an idealistic superscalar processor. All of the figures for these results, Figures 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, and 7.8, are of a similar format. Each gives performance as the speedup of the multiscalar and superscalar processors over the baseline processor shown as stacked bar graphs. Each of the figures has a multiscalar side (on the left) and a superscalar side (on the right).

The superscalar side of each figure (on the right) gives three stacked bars, corresponding to instruction window structures of a 64 entry in-flight queue and a 16 entry on-deck queue (given as **s64x16**), a 128 entry in-flight queue and a 32 entry on-deck queue (given as **s128x32**), or a 256 entry in-flight queue and a 64 entry on-deck queue (given as **s256x64**). (Definitions for the terms in-flight queue and on-deck queue, as related to the instruction window, may be found in Section 4.5.3.3 of Chapter 4.) The stack on each bar represents the increases in issue width for the out-of-order processor going from 4-wide (given as **processor=oo.u4**) to 8-wide (given as **processor=oo.u8**) as well as from 8-wide to 16-wide (given as **processor=oo.u16**).

If the multiscalar processor runs a single task per processing unit, then the multiscalar side of the figure (on the left) gives three stacked bars, corresponding to multiscalar processors with 4 (given as **m4**), 8 (given as **m8**), and 16 (given as **m16**) processing units. Each processing unit has an instruction window structure of a 16 entry in-flight queue and a 8 entry on-deck queue. The stack on each bar represents increases in issue-width for the out-of-order processing units going from 1-wide (given as **proc unit=oo.u1**) to 2-wide (given as **proc unit=oo.u2**) and from 2-wide to 4-wide (given as **proc unit=oo.u4**).

If the multiscalar processor runs multiple tasks per processing unit, then the multiscalar side of the figure (on the left) gives three stacked bars, corresponding to a multiscalar processor with 4 processing units running 1 task (given as **m4**), 2 tasks (given as **m4x2.all**), or 4 tasks (given as **m4x4.all**) all-at-once. (Note, the m4 configuration is the same as for a single task per processing unit.) Each processing unit has an instruction window structure of a 16 entry in-flight queue and a 8 entry on-deck queue. The stack on each bar represents increases in issue-width for the out-of-order processing units going from 1-wide (given as **proc unit=oo.u1**) to 2-wide (given as **proc unit=oo.u2**) and from 2-wide to 4-wide (given as **proc unit=oo.u4**).

In addition, underneath each bar graph is a table that provides the percentage differences between the multiscalar and superscalar speedups. A positive percentage indicates that a multiscalar processor has a bigger speedup than a superscalar processor by the amount given. (These table cells are shaded, and their values are shown in bold typeface to make them easier to distinguish.) A negative percentage indicates a multiscalar processor has a smaller speedup by the amount given. (These table cells are not shaded, and their values are shown in normal typeface.) The table breaks down all of the stacks in the figure so that it is possible to compare any superscalar instruction window and processor issue width configuration with any multiscalar task and processing unit issue width configuration.

7.3.4.1 Conservative Multiscalar versus Superscalar

Before describing the presentation of these results, it is worth reiterating that the multiscalar processors considered in this part use conservative latency characteristics with respect to the data supply. In particular, the register propagation delay is 2 cycles for the first "hop" with 1 cycle for each "hop" thereafter, and the data cache hit latency is 3 cycles.

This performance comparison investigates two different type of conservative multiscalar processors. The first type is configured with processing units that each run a single task (one-at-a-time). The second type is configured with processing units that each run multiple tasks all-at-once. The results for the multiple tasks per processing unit configurations are described after the results for the single task per processing unit configurations. Furthermore, the results for both the SPEC CINT95 benchmarks and the SPEC CFP95 benchmarks are presented. An analysis of these results is not provided here, but instead after the next section.

Figure 7.1 compares the performance in terms of speedup over the baseline processor of realistic multiscalar processors with conservative latency assumptions running a single task per processing unit (one-at-a-time) and idealistic superscalar processors for the SPEC CINT95 benchmarks. Likewise, Figure 7.2 compares the performance in terms of speedup over the baseline processor of realistic multiscalar processors with conservative latency assumptions running multiple tasks per processing unit all-at-once and idealistic superscalar processors for the SPEC CINT95 benchmarks.

Figure 7.3 compares the performance in terms of speedup over the baseline processor of realistic multiscalar processors with conservative latency assumptions running a single task per processing unit (one-at-a-time) and idealistic superscalar processors for the SPEC CFP95 benchmarks. Likewise, Figure 7.4 compares the performance in terms of speedup over the baseline processor of realistic multiscalar processors with conservative latency assumptions running multiple tasks per processing unit all-at-once and idealistic superscalar processors for the SPEC CFP95 benchmarks.

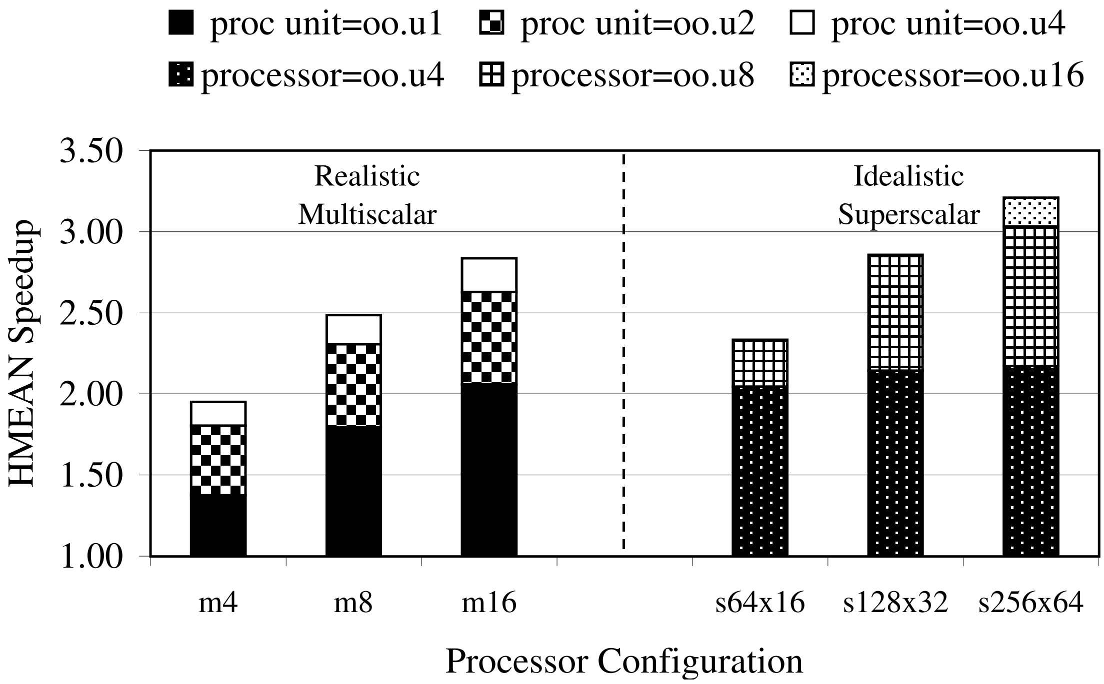
7.3.4.2 Aggressive Multiscalar versus Superscalar

As above, it is worth reiterating that the multiscalar processors considered in this part use aggressive latency characteristics with respect the data supply before describing the presentation of these results. To be specific, the register propagation delay is 1 cycle for each "hop", and the data cache hit latency is 2 cycles.

This performance comparison investigates two different type of conservative multiscalar processors. The first type is configured with processing units that each run a single task (one-at-a-time). The second type is configured with processing units that each run multiple tasks all-at-once. The results for the multiple tasks per processing unit configurations are described after the results for the single task per processing unit configurations. Furthermore, the results for both the SPEC CINT95 benchmarks and the SPEC CFP95 benchmarks are presented. A specific and general analysis of these results and those of the previous section are provided in the next two sections.

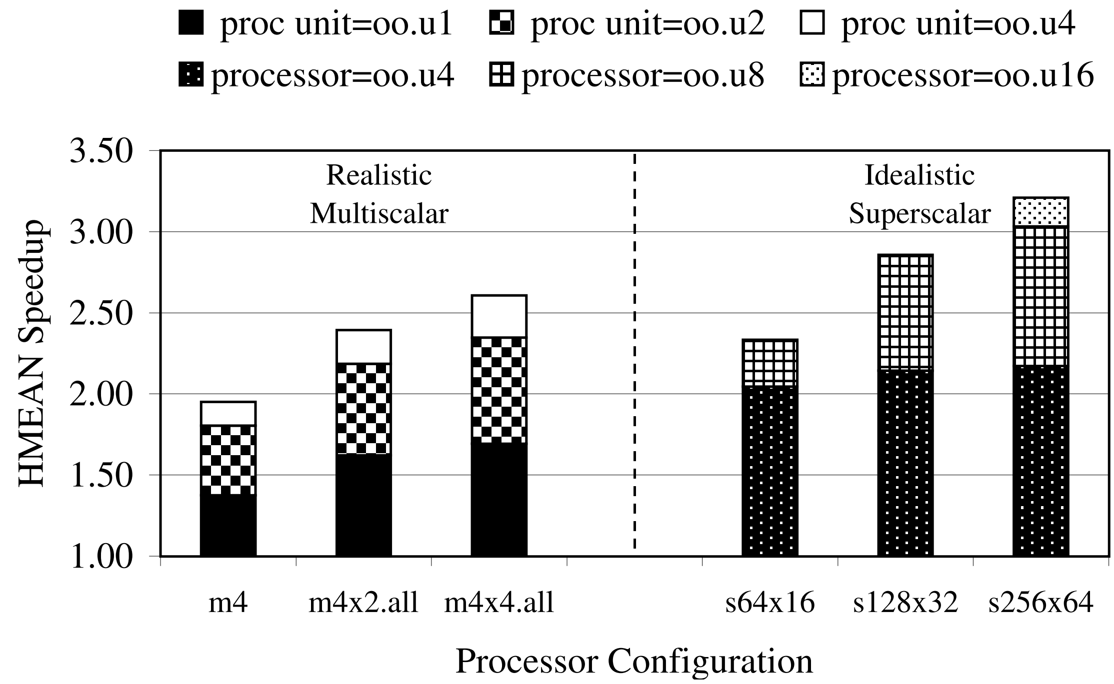
Figure 7.5 compares the performance in terms of speedup over the baseline processor of realistic multiscalar processors with aggressive latency assumptions running a single task per processing unit (one-at-a-time) and idealistic superscalar processors for the SPEC CINT95 benchmarks. Figure 7.6 compares the performance in terms of speedup over the baseline processor of realistic multiscalar processors with aggressive latency assumptions running multiple tasks per processing unit all-at-once and idealistic superscalar processors for the SPEC CINT95 benchmarks.

Figure 7.7 compares the performance in terms of speedup over the baseline processor of realistic multiscalar processors with aggressive latency assumptions running a single task per processing unit (one-at-a-time) and idealistic superscalar processors for the SPEC CFP95 benchmarks. Figure 7.8 compares the performance in terms of speedup over the baseline processor of realistic multiscalar processors with aggressive latency assumptions running multiple tasks per processing unit all-at-once and idealistic superscalar processors for the SPEC CFP95 benchmarks.



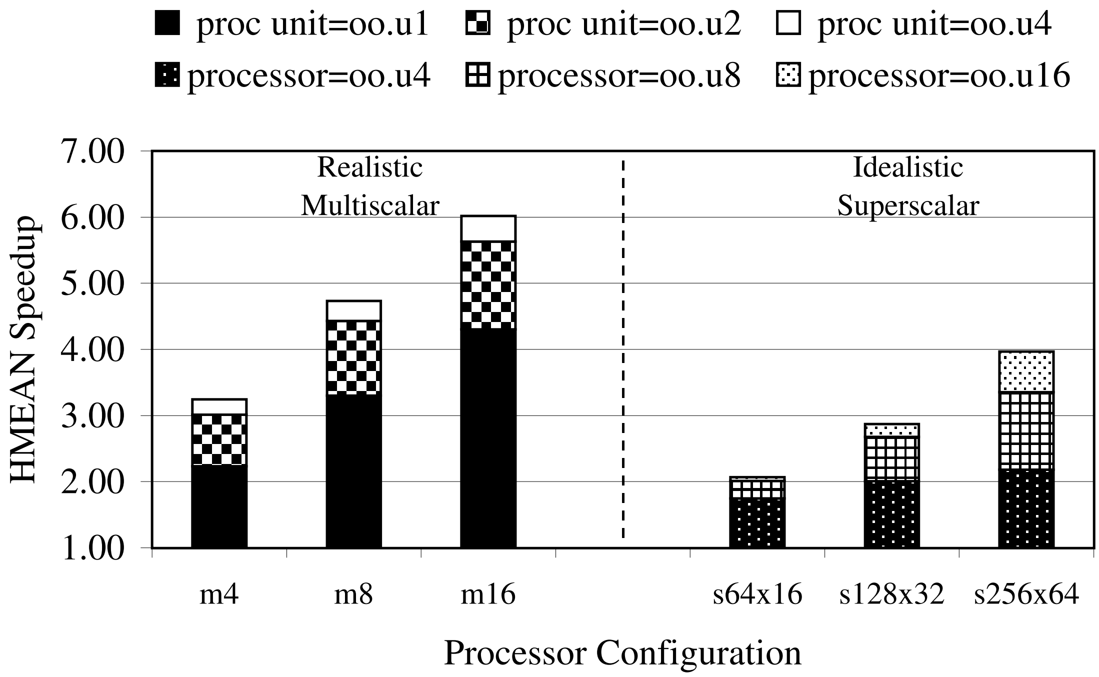
		m4			m8			m16		
		oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4
s64x16	oo.u4	-48.7%	-13.3%	-4.8%	-13.9%	**11.4%**	**17.7%**	**0.7%**	**22.2%**	**27.9%**
	oo.u8	-69.8%	-29.4%	-19.6%	-30.0%	-1.2%	**6.0%**	-13.4%	**11.2%**	**17.7%**
	oo.u16	-69.8%	-29.4%	-19.6%	-30.0%	-1.2%	**6.0%**	-13.4%	**11.2%**	**17.7%**
s128x32	oo.u4	-56.0%	-18.8%	-9.9%	-19.4%	**7.1%**	**13.7%**	-4.2%	**18.4%**	**24.4%**
	oo.u8	-107.8%	-58.4%	-46.4%	-59.1%	-23.9%	-15.0%	-38.8%	-8.7%	-0.7%
	oo.u16	-107.8%	-58.4%	-46.4%	-59.1%	-23.9%	-15.0%	-38.8%	-8.7%	-0.7%
s256x64	oo.u4	-58.0%	-20.4%	-11.3%	-20.9%	**5.9%**	**12.6%**	-5.5%	**17.4%**	**23.5%**
	oo.u8	-120.6%	-68.1%	-55.4%	-68.9%	-31.4%	-22.0%	-47.3%	-15.4%	-6.9%
	oo.u16	-133.5%	-77.9%	-64.5%	-78.8%	-39.2%	-29.2%	-56.0%	-22.2%	-13.2%

Figure 7.1: Comparison of conservative multiscalar processors running a single task per processing unit and superscalar processors for the SPEC CINT95 benchmarks.



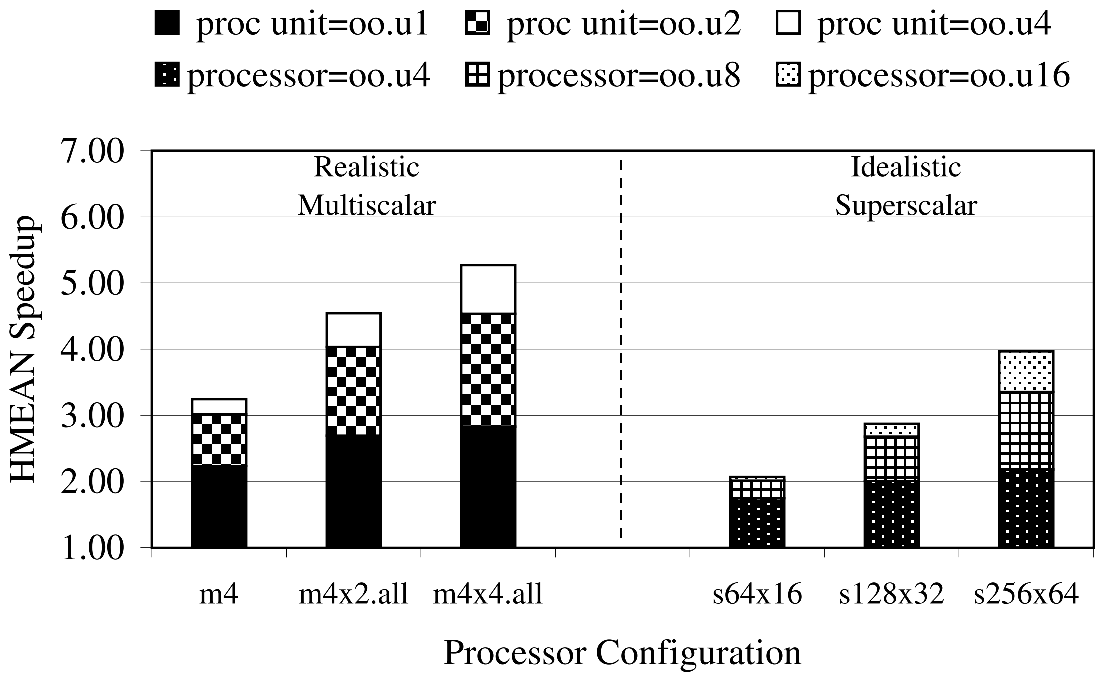
		m4			m4x2.all			m4x4.all		
		oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4
s64x16	oo.u4	-48.7%	-13.3%	-4.8%	-26.0%	**6.4%**	**14.6%**	-20.8%	**12.9%**	**21.6%**
	oo.u8	-69.8%	-29.4%	-19.6%	-43.9%	-6.8%	**2.5%**	-37.9%	**0.6%**	**10.5%**
	oo.u16	-69.8%	-29.4%	-19.6%	-43.9%	-6.8%	**2.5%**	-37.9%	**0.6%**	**10.5%**
s128x32	oo.u4	-56.0%	-18.8%	-9.9%	-32.1%	**1.9%**	**10.4%**	-26.6%	**8.7%**	**17.8%**
	oo.u8	-107.8%	-58.4%	-46.4%	-76.1%	-30.8%	-19.4%	-68.8%	-21.7%	-9.5%
	oo.u16	-107.8%	-58.4%	-46.4%	-76.1%	-30.8%	-19.4%	-68.8%	-21.7%	-9.5%
s256x64	oo.u4	-58.0%	-20.4%	-11.3%	-33.8%	**0.6%**	**9.3%**	-28.3%	**7.5%**	**16.7%**
	oo.u8	-120.6%	-68.1%	-55.4%	-86.9%	-38.8%	-26.7%	-79.1%	-29.1%	-16.3%
	oo.u16	-133.5%	-77.9%	-64.5%	-97.9%	-46.9%	-34.1%	-89.6%	-36.7%	-23.1%

Figure 7.2: Comparison of conservative multiscalar processors running multiple tasks per processing unit all-at-once and superscalar processors for the SPEC CINT95 benchmarks.



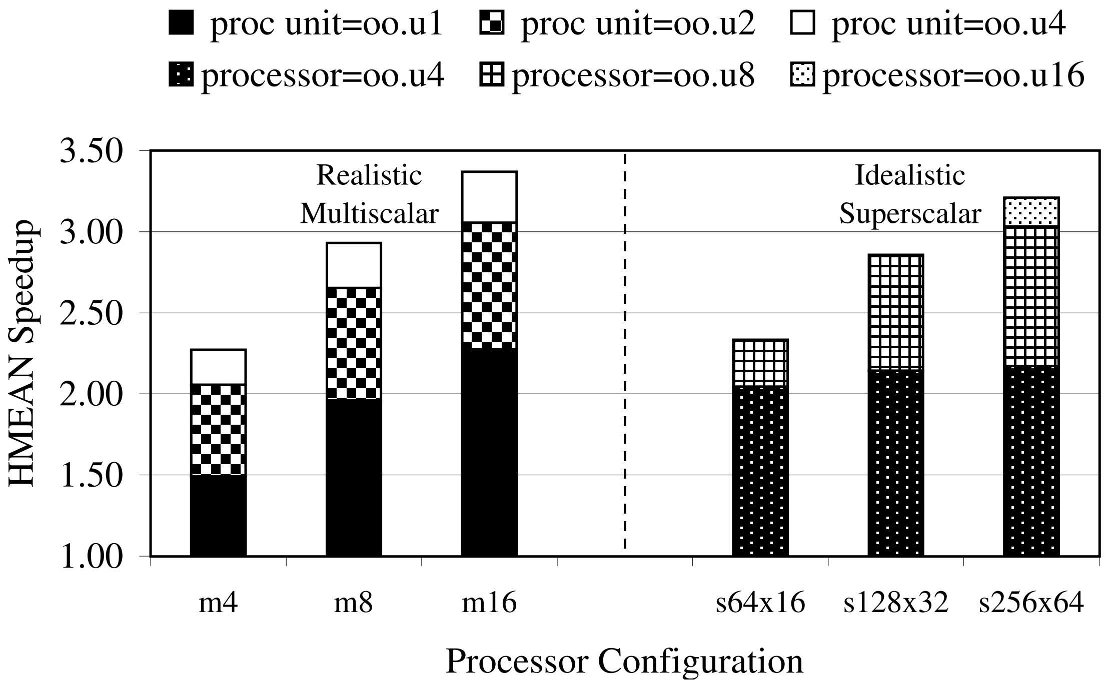
		m4			m8			m16		
		oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4
s64x16	oo.u4	**21.7%**	**42.1%**	**46.2%**	**47.1%**	**60.6%**	**63.1%**	**59.4%**	**69.0%**	**71.0%**
	oo.u8	**9.9%**	**33.3%**	**38.1%**	**39.1%**	**54.7%**	**57.5%**	**53.3%**	**64.3%**	**66.6%**
	oo.u16	**7.3%**	**31.4%**	**36.3%**	**37.4%**	**53.4%**	**56.3%**	**51.9%**	**63.3%**	**65.6%**
s128x32	oo.u4	**10.1%**	**33.4%**	**38.2%**	**39.2%**	**54.7%**	**57.6%**	**53.4%**	**64.4%**	**66.7%**
	oo.u8	-20.0%	**11.2%**	**17.5%**	**18.9%**	**39.6%**	**43.4%**	**37.8%**	**52.5%**	**55.5%**
	oo.u16	-28.8%	**4.7%**	**11.5%**	**13.0%**	**35.2%**	**39.3%**	**33.2%**	**49.0%**	**52.3%**
s256x64	oo.u4	**2.1%**	**27.6%**	**32.7%**	**33.9%**	**50.7%**	**53.9%**	**49.2%**	**61.2%**	**63.7%**
	oo.u8	-50.3%	-11.3%	-3.3%	-1.6%	**24.3%**	**29.1%**	**22.0%**	**40.5%**	**44.3%**
	oo.u16	-77.8%	-31.6%	-22.2%	-20.1%	**10.5%**	**16.2%**	**7.8%**	**29.6%**	**34.1%**

Figure 7.3: Comparison of conservative multiscalar processors running a single task per processing unit and superscalar processors for the SPEC CFP95 benchmarks.



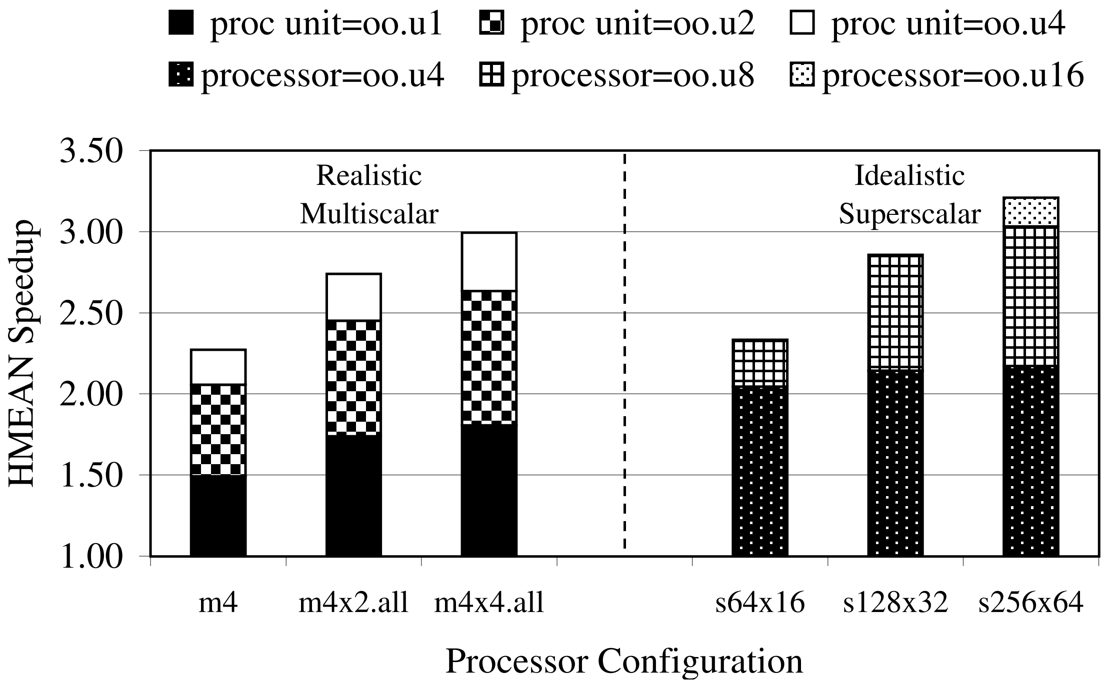
		m4			m4x2.all			m4x4.all		
		oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4
s64x16	oo.u4	**21.7%**	**42.1%**	**46.2%**	**35.2%**	**56.7%**	**61.6%**	**38.3%**	**61.5%**	**66.9%**
	oo.u8	**9.9%**	**33.3%**	**38.1%**	**25.4%**	**50.1%**	**55.8%**	**29.0%**	**55.7%**	**61.9%**
	oo.u16	**7.3%**	**31.4%**	**36.3%**	**23.2%**	**48.7%**	**54.5%**	**27.0%**	**54.4%**	**60.8%**
s128x32	oo.u4	**10.1%**	**33.4%**	**38.2%**	**25.5%**	**50.2%**	**55.9%**	**29.1%**	**55.8%**	**61.9%**
	oo.u8	-20.0%	**11.2%**	**17.5%**	**0.6%**	**33.6%**	**41.1%**	**5.4%**	**41.0%**	**49.2%**
	oo.u16	-28.8%	**4.7%**	**11.5%**	-6.7%	**28.7%**	**36.8%**	-1.5%	**36.7%**	**45.5%**
s256x64	oo.u4	**2.1%**	**27.6%**	**32.7%**	**18.9%**	**45.8%**	**52.0%**	**22.9%**	**51.9%**	**58.6%**
	oo.u8	-50.3%	-11.3%	-3.3%	-24.5%	**16.8%**	**26.2%**	-18.5%	**26.1%**	**36.4%**
	oo.u16	-77.8%	-31.6%	-22.2%	-47.3%	**1.6%**	**12.7%**	-40.2%	**12.5%**	**24.7%**

Figure 7.4: Comparison of conservative multiscalar processors running multiple tasks per processing unit all-at-once and superscalar processors for SPEC the CFP95 benchmarks.



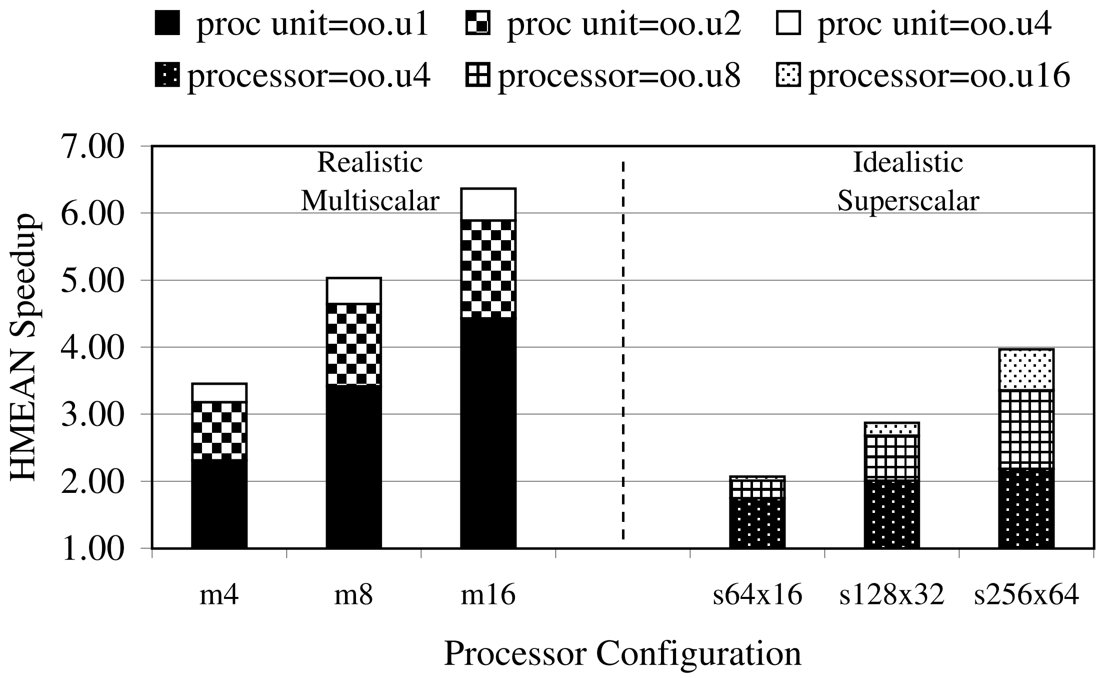
		m4			m8			m16		
		oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4
s64x16	oo.u4	-36.9%	**0.6%**	**10.0%**	-4.3%	**22.9%**	**30.2%**	**10.1%**	**33.1%**	**39.3%**
	oo.u8	-56.3%	-13.6%	-2.7%	-19.1%	**12.0%**	**20.3%**	-2.7%	**23.6%**	**30.7%**
	oo.u16	-56.3%	-13.6%	-2.7%	-19.1%	**12.0%**	**20.3%**	-2.7%	**23.6%**	**30.7%**
s128x32	oo.u4	-43.5%	-4.3%	**5.7%**	-9.4%	**19.2%**	**26.8%**	**5.7%**	**29.8%**	**36.4%**
	oo.u8	-91.3%	-39.0%	-25.7%	-45.8%	-7.7%	**2.5%**	-25.6%	**6.5%**	**15.2%**
	oo.u16	-91.3%	-39.0%	-25.7%	-45.8%	-7.7%	**2.5%**	-25.6%	**6.5%**	**15.2%**
s256x64	oo.u4	-45.4%	-5.6%	**4.4%**	-10.8%	**18.1%**	**25.9%**	**4.5%**	**28.9%**	**35.5%**
	oo.u8	-103.0%	-47.5%	-33.4%	-54.7%	-14.3%	-3.5%	-33.3%	**0.7%**	**10.0%**
	oo.u16	-114.9%	-56.2%	-41.3%	-63.8%	-21.0%	-9.6%	-41.2%	-5.1%	**4.7%**

Figure 7.5: Comparison of aggressive multiscalar processors running a single task per processing unit and superscalar processors for the SPEC CINT95 benchmarks.



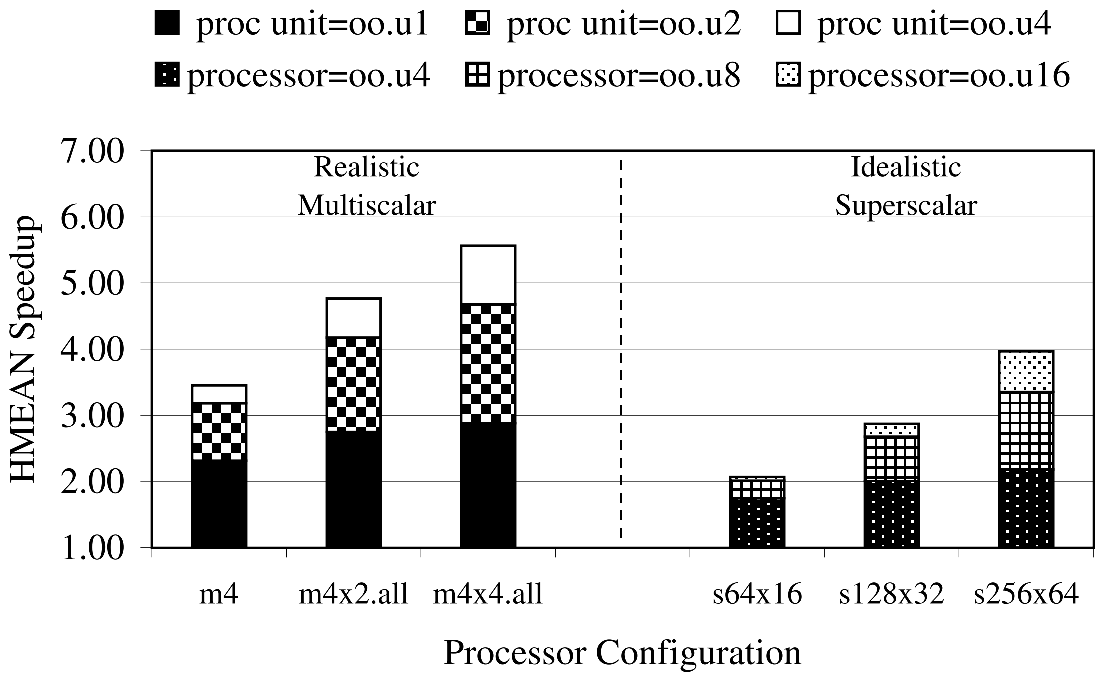
		m4			m4x2.all			m4x4.all		
		oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4
s64x16	oo.u4	-36.9%	**0.6%**	**10.0%**	-17.5%	**16.6%**	**25.4%**	-13.2%	**22.4%**	**31.7%**
	oo.u8	-56.3%	-13.6%	-2.7%	-34.2%	**4.8%**	**14.8%**	-29.3%	**11.4%**	**22.0%**
	oo.u16	-56.3%	-13.6%	-2.7%	-34.2%	**4.8%**	**14.8%**	-29.3%	**11.4%**	**22.0%**
s128x32	oo.u4	-43.5%	-4.3%	**5.7%**	-23.2%	**12.5%**	**21.8%**	-18.7%	**18.6%**	**28.4%**
	oo.u8	-91.3%	-39.0%	-25.7%	-64.2%	-16.5%	-4.3%	-58.2%	-8.5%	**4.6%**
	oo.u16	-91.3%	-39.0%	-25.7%	-64.2%	-16.5%	-4.3%	-58.2%	-8.5%	**4.6%**
s256x64	oo.u4	-45.4%	-5.6%	**4.4%**	-24.8%	**11.4%**	**20.7%**	-20.3%	**17.6%**	**27.5%**
	oo.u8	-103.0%	-47.5%	-33.4%	-74.3%	-23.7%	-10.7%	-67.9%	-15.1%	-1.3%
	oo.u16	-114.9%	-56.2%	-41.3%	-84.5%	-31.0%	-17.2%	-77.8%	-21.9%	-7.2%

Figure 7.6: Comparison of aggressive multiscalar processors running multiple tasks per processing unit all-at-once and superscalar processors for the SPEC CINT95 benchmarks.



		m4			m8			m16		
		oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4
s64x16	oo.u4	**24.5%**	**45.1%**	**49.4%**	**48.9%**	**62.4%**	**65.3%**	**60.6%**	**70.4%**	**72.6%**
	oo.u8	**13.1%**	**36.8%**	**41.8%**	**41.2%**	**56.7%**	**60.0%**	**54.6%**	**65.9%**	**68.4%**
	oo.u16	**10.6%**	**35.0%**	**40.1%**	**39.5%**	**55.5%**	**58.9%**	**53.3%**	**64.9%**	**67.5%**
s128x32	oo.u4	**13.2%**	**36.9%**	**41.9%**	**41.3%**	**56.8%**	**60.1%**	**54.7%**	**65.9%**	**68.5%**
	oo.u8	-15.8%	**15.8%**	**22.4%**	**21.7%**	**42.4%**	**46.8%**	**39.6%**	**54.5%**	**57.9%**
	oo.u16	-24.2%	**9.7%**	**16.8%**	**16.0%**	**38.2%**	**42.9%**	**35.2%**	**51.2%**	**54.9%**
s256x64	oo.u4	**5.6%**	**31.4%**	**36.8%**	**36.1%**	**53.0%**	**56.6%**	**50.7%**	**62.9%**	**65.7%**
	oo.u8	-45.0%	-5.4%	**2.9%**	**1.9%**	**27.8%**	**33.3%**	**24.3%**	**43.1%**	**47.3%**
	oo.u16	-71.6%	-24.7%	-14.9%	-16.0%	**14.6%**	**21.1%**	**10.5%**	**32.6%**	**37.7%**

Figure 7.7: Comparison of aggressive multiscalar processors running a single task per processing unit and superscalar processors for the SPEC CFP95 benchmarks.



		m4			m4x2.all			m4x4.all		
		oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4	oo.u1	oo.u2	oo.u4
s64x16	oo.u4	**24.5%**	**45.1%**	**49.4%**	**36.5%**	**58.2%**	**63.4%**	**39.4%**	**62.7%**	**68.6%**
	oo.u8	**13.1%**	**36.8%**	**41.8%**	**26.9%**	**51.9%**	**57.8%**	**30.2%**	**57.0%**	**63.9%**
	oo.u16	**10.6%**	**35.0%**	**40.1%**	**24.8%**	**50.5%**	**56.6%**	**28.2%**	**55.8%**	**62.8%**
s128x32	oo.u4	**13.2%**	**36.9%**	**41.9%**	**27.0%**	**52.0%**	**57.9%**	**30.4%**	**57.1%**	**63.9%**
	oo.u8	-15.8%	**15.8%**	**22.4%**	**2.6%**	**35.9%**	**43.8%**	**7.1%**	**42.8%**	**51.9%**
	oo.u16	-24.2%	**9.7%**	**16.8%**	-4.5%	**31.2%**	**39.7%**	**0.3%**	**38.6%**	**48.4%**
s256x64	oo.u4	**5.6%**	**31.4%**	**36.8%**	**20.5%**	**47.7%**	**54.2%**	**24.2%**	**53.3%**	**60.8%**
	oo.u8	-45.0%	-5.4%	**2.9%**	-22.0%	**19.7%**	**29.6%**	-16.4%	**28.3%**	**39.7%**
	oo.u16	-71.6%	-24.7%	-14.9%	-44.4%	**5.0%**	**16.8%**	-37.7%	**15.2%**	**28.7%**

Figure 7.8: Comparison of aggressive multiscalar processors running multiple tasks per processing unit all-at-once and superscalar processors for the SPEC CFP95 benchmarks.

7.3.4.3 Specific Analysis

The collection of results presented in the previous two sections indicate the same overall characteristics comparing multiscalar and superscalar processors, regardless of whether conservative or aggressive latency assumptions are used. That is, the results indicate that multiscalar processors outperform superscalar processors over nearly the entire range of configurations for floating point programs, but over only a selected range of configurations for integer programs. As to be expected, the coverage of these ranges increases as the number of processing units increases when running a single task per processing unit (one-at-a-time) and as the number of tasks per processing unit increases when running multiple tasks per processing unit all-at-once.

For the integer programs, several specific configurations offer equivalent and often significantly better performance than all superscalar configurations. A multiscalar processor configured as 16 processing units with 2-wide or 4-wide issue running 1 task per processing unit provides better performance than nearly any superscalar processor configuration, with the aggressive latency assumptions. Yet even with the conservative latency assumptions, this multiscalar configuration provides better performance than most superscalar processor configurations. A multiscalar processor configured as 4 processing units running 4 tasks per processing unit offers similar performance to the 16 processing unit configuration but does so for what is anticipated to be a much lower cost. Likewise, multiscalar processors configured as 8 processing units running 1 task per processing unit or as 4 processing units running 2 tasks per processing unit provide comparable alternatives.

For the floating point programs, almost every configuration offers significantly better performance than all superscalar configurations. Even a multiscalar processor configured as 4 processing units running 1 task per processing unit provides better performance than most superscalar processor configurations. Furthermore, this performance advantage holds for both the conservative and the aggressive latency assumptions. The performance advantages for multiscalar processors configured as 8 or 16 processing units running 1 task per processing unit and as 4 processing units running 2 tasks or 4 tasks per processing unit are extremely high, in many case 50% or more, even when compared to the most ambitious superscalar processor configurations. Yet, because these advantages for multiscalar processors over superscalar processors are found over almost the entire cost range, there are many alternatives to provide these levels of performance.

7.3.4.4 General Analysis

Comparing the situations where the speedup is bigger for multiscalar processors than for superscalar processors, it appears that the same patterns (seen by examining the tables beneath the bar graphs) exist regardless of whether the multiscalar processors run a single task per processing unit or multiple tasks per processing or use conservative or aggressive latency assumptions. That is, it always appears to be the same configurations that perform better. Of course, as already described, the pattern and number of such configurations are quite different for the SPEC CINT95 benchmarks as compared to the SPEC CFP95 benchmarks. Specifically, there is a narrow range of cases with performance advantages for integer as compared to a wide range of cases with performance advantages for floating point.

Nevertheless, even though the overall pattern and number of configurations with better performance are consistent (aside from the integer versus floating point benchmark differences), it is important to point out that the percentage differences in performance are bigger for multiscalar processors configured to run a single task per processing unit than for those configured to run multiple tasks per processing unit. That is, when performance advantages exist, the single task multiscalar configurations provide bigger performance improvements over the superscalar configurations, than the multiple task multiscalar configurations do. These differences range from 5% to 15% for the SPEC CINT95 benchmarks and from 5% to 50% for the SPEC CFP95 benchmarks across both conservative and aggressive latency characteristics.

Likewise, the percentage differences in performance are bigger for multiscalar processors configured with aggressive latency assumptions than for those configured with conservative latency assumptions. Thus, as expected, when performance advantages exist, multiscalar configurations with aggressive latency assumptions provide bigger performance improvements over superscalar configurations than those with conservative latency assumptions. For the SPEC CINT95 benchmarks the range of differences for conservative versus aggressive configurations are roughly the same as those for single task versus multiple tasks per processing unit, about 10% to 15%. However, for the SPEC CFP95 benchmarks the range of differences are much smaller than those for single task versus multiple tasks per processing unit, only about 0% to 5%.

Considering the overall patterns as well as the percentage differences in performance (related to single task versus multiple tasks per processing unit and conservative versus aggressive latency characteristics), it is possible to draw a general conclusion with regard to these performance factors. It appears that the ability of a multiscalar processor to provide better performance than a superscalar processor is primarily influenced by the number of tasks in concurrent execution on the processor and is secondarily influenced by the overall issue width of the processor. The evidence upon which these conclusions are drawn is twofold.

First, processor configurations with the highest number of tasks in concurrent execution provide the highest performance across both the integer and floating point benchmarks. Furthermore, as this observation would suggest, for a processor configuration with a fixed number of processing units, performance decreases as the number of tasks in concurrent execution decreases. For instance, the **m16** and the **m4x4.all** configurations, those with the highest performance, have the highest number of tasks in current execution, 16; these are followed by the **m8** and the **m4x2.all** configurations, and then the **m4** configuration. Nonetheless, it is clear that running the same number of tasks with more processing units provides higher performance.

Second, processor configurations with more overall issue width but less tasks in concurrent execution usually do not provide higher performance than processor configurations with less overall issue width but more tasks in concurrent execution. Nonetheless, for a processor configuration with a fixed number of processing units and tasks, higher performance comes with higher processor (or actually processing unit) issue width. For instance, the **m4x4.all oo.u4** configuration with 16 concurrent tasks and 16 overall issue outperforms the **m8 oo.u4** configuration with 8 concurrent tasks and 32 overall issue. Similarly, the **m4x2.all oo.u2** configuration with 8 concurrent tasks and 8 overall issue outperforms the **m4 oo.u4** configuration with 4 concurrent tasks and 16 overall issue.

7.4 Putting It All Together

The evidence presented in the previous section might lead one to conclude that a realistic multiscalar processor is only likely to attain better performance than an idealistic superscalar processor when it provides a "better" instruction window. Indeed, the multiscalar instruction window must be much better than the superscalar instruction window because the multiscalar processor, modeled in a realistic manner, must overcome overheads that the superscalar processor, modeled in an idealistic manner, can simply ignore.

These overheads include the obvious losses due to instruction count increases as well as the subtle losses due to the lack of scheduling flexibility that decreases instruction-level parallel execution. Thus, the realistic multiscalar processors pay a price in lost performance from instruction-level parallelism that the idealistic superscalar processors do not. Yet, the price paid in terms of instruction-level parallelism is compounded by the fact that it is related to another source of lost performance attributed to the same differences between realistic and idealistic designs.

It is worth recalling from the discussion at the beginning of this chapter, that there is a key factor in the performance equation that is not accounted for in the performance comparison results, the time per cycle or clock frequency. What is significant is that the design choices that are responsible for the losses in terms of instruction-level parallelism are the same ones that are responsible for the gains in terms of clock frequency. As a result, the realistic multiscalar designs pay their price two times over, while the idealistic superscalar designs do not pay at all.

Unfortunately, these two discrepancies with respect to the performance that arise due to the differences between the realistic and idealistic nature of the designs being compared are difficult (if not impossible) to quantify. There is, however, some hope of putting this performance comparison in perspective. Though the experimental framework used in this work cannot measure these two sources of performance loss, it can be used to consider the possible scenarios in which their contributions might play a key role in whether multiscalar processors can provide performance advantages over superscalar processors.

Using the percentage differences between multiscalar and superscalar performance measured without taking these losses into account, it is possible to divide them up into one of two groups: (1) those where the measured results show multiscalar processors have a performance advantage over superscalar processors and (2) those where the measure results do not show a performance advantage. The first group corresponds to the shaded cells of the percentage difference tables given earlier in Figures 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, and 7.8. The second group corresponds to the non-shaded cells of the percentage difference tables in those same figures.

For the first group of results, it is clear that the multiscalar processors have a performance advantage compared to the superscalar processors, in spite of the fact that there are performance losses not accounted for in the performance comparison. For example, in Figure 7.5, a multiscalar processor with 16 4-wide out-of-order processing units using aggressive latency assumptions has a 4.7% performance advantage compared to a superscalar processor with 16-wide out-of-order issue for an instruction window using a 256 in-flight queue and a 64 entry on-deck queue. The advantage of this realistic multiscalar design is of even more significance if it is coupled with the fact that a realistic superscalar design would have degraded instruction-level parallelism and/or clock frequency compared to the idealistic superscalar design reported.

For the second group of results, it is unclear whether the multiscalar processors have a performance advantage compared to the superscalar processors. Because there are performance losses not accounted for in the performance comparison, it can only be said that their contribution must be comparable to the percentage difference in performance between the multiscalar and superscalar designs in order to reach the break-even point. For instance, in Figure 7.1, a multiscalar processor with 16 4-wide out-of-order processing units using conservative latency assumptions has a 13.2% performance disadvantage compared to a superscalar processor with 16-wide out-of-order issue for an instruction window using a 256 in-flight queue and a 64 entry on-deck queue. Thus, a realistic superscalar design must have degraded instruction-level parallelism and/or clock frequency of at least that amount compared to the idealistic superscalar design reported in order for a realistic multiscalar design to have an advantage.

As the examination of these two cases shows, it is important to use care in considering the results of this performance comparison before drawing any conclusions. Using the percentage differences as break-even points is probably the proper means to gauge how the alternative multiscalar approach fairs against the conventional superscalar approach. Yet, the value of the break-even point at which it is compelling to consider a multiscalar design over a superscalar design remains unknown. It most certainly depends on the application programs as well as the implementation technology for which a processor is targeted. In fact, not only might the percent differences need to favor multiscalar over superscalar (that is, be positive), it may be the case that the magnitude of these percentage differences must be of sufficient value to overcome the inertia associated with the fact that superscalar processors are a proven technology while multiscalar processors are only a potential technology. Whatever the criteria might be, it is beyond the bounds of this work to endeavor to determine them.

7.5 Summary

This chapter provided a performance comparison, relative to a baseline 1-wide out-of-order issue processor, between realistic multiscalar processors and idealistic superscalar processors. An idealistic superscalar design was used for this purpose because existing realistic superscalar designs did not possess capabilities for extracting instruction-level parallelism (in terms of issue-width, instruction window size, and clock frequency) that were comparable to those expected for multiscalar designs like those studied in this thesis. The key difference between the realistic and idealistic designs was that the performance of the realistic design took into account all of the overheads required to accommodate implementation technology, while the idealistic design did not.

Due to these differences, there might be discrepancies with respect to the clock frequency factor that were not accounted for in the performance comparison. Nevertheless, the multiscalar processors and superscalar processors were comparable in terms of their instruction and data memory systems as well as their overall processing capabilities. Still, the instruction windows from which instruction-level parallelism was extracted were quite different because of the fundamental differences in the two paradigms. That is, the superscalar processors used a single centralized window whose sizes varied in terms of number of instructions, while the multiscalar processors used multiple decentralized windows whose sizes varied in terms of number of tasks. Furthermore, the multiscalar processors used in the performance comparison were evaluated with two different data latency configurations, one conservative and the other aggressive.

Overall, the collection of results indicated that nearly all realistic multiscalar processor configurations (from most to least powerful) outperform idealistic superscalar processor configurations for the SPEC CFP95 programs. However, only the most powerful realistic multiscalar processor configurations outperform idealistic superscalar processor configurations for the SPEC CINT95 programs. Nonetheless, it was clear in all cases that multiscalar performance increased relative to superscalar performance as the number of processing units increased when running a single task per processing unit (one-at-a-time) and as the number of tasks per processing unit increased when running multiple tasks per processing unit all-at-once. As a result, it was concluded that the ability of a multiscalar processor to provide better performance than a superscalar processor was primarily influenced by the number of tasks in concurrent execution on the processor and was secondarily influenced by the overall issue width of the processor.

Chapter 8

Conclusion

Future processor designs intended to meet the ever-growing demand for performance via instruction-level parallelism and clock frequency must do so within the constraints of future implementation technology and within the limits of practicable implementation costs. The forecasted trends in future implementation technology are straightforward. In the past, on-chip communication was cheap, but transistors were too scarce to explore high degrees of instruction-level parallelism. In the future, transistors should be plentiful, but on-chip communication between these transistors could be expensive.

The multiscalar paradigm offers a novel approach to extract instruction-level parallelism (ILP) from ordinary serial programs that is well-matched to the constraints of future implementation technology. A multiscalar processor, an implementation of this paradigm, exposes a very large window of instructions by splitting the dynamic instruction sequence of a serial program – using hardware, software, or some combination thereof – into a collection of control and/or data dependent regions. It exploits instruction-level parallelism using a decentralized hardware organization that executes these regions in parallel on multiple processing units.

8.1 Contributions

The work presented in this thesis has made three major contributions in terms of advancing knowledge of the multiscalar paradigm and furnishing valuable insight for future computing paradigms and their microarchitectures. First, it has developed a simulation infrastructure (to complement the compiler infrastructure developed in tandem [97]). Second, it has performed the first comprehensive study in terms of design and evaluation of a multiscalar processor as well as its comparison to a superscalar processor. Third, it has provided a baseline of data to guide future work on multiscalar processors as well as for the continued evolution of superscalar processors and multiprocessors. These three contributions were not independent of one another, but instead were the dependent parts of a directed study. While this study focused on multiscalar processors in particular, many of the concepts and insights are applicable to processors in general.

This thesis considered a multiscalar processor that relied on a compiler to convey static information to the processor to support the dynamic multiscalar execution model – other approaches are possible. To conduct a methodical investigation, this work focused on the three most significant aspects of a multiscalar processor: instruction and data processing, instruction supply, and data supply. Detailed design descriptions and experimental evaluations were provided for each aspect, identifying the impact of each from the standpoint of its individual performance as well as its contribution to overall performance. In addition, a performance comparison of realistic multiscalar processors relative to idealistic superscalar processors was provided to give an impression of how this alternative approach might fair against a well-known conventional approach.

8.2 Instruction and Data Processing

The amount of instruction-level parallelism that a multiscalar processor is able to expose and exploit is influenced by the number and/or size of the tasks that make up its instruction window as well as the interaction of that instruction window with the processing units that execute the instructions within it. The negative impact on performance of this interaction may be attributed to the fact that some of the processing phases performed during the lifetime of a task must be ordered among tasks. However, the extent to which this ordering is able to limit performance is determined by various inter-task and intra-task aspects of the processing unit design.

8.2.1 Inter-Task Aspect

In studying the inter-task aspect of processing unit design, one-to-one and many-to-one task assignment were identified as the two key alternatives that must be considered in the design of a multiscalar processor. As related to many-to-one task assignment, this thesis focused on two issues: mapping policy and running policy. The mapping policy was defined as the means by which the processor dictates how tasks are assigned to processing units; while the running policy was defined as the means by which the processor dictates how many tasks a processing unit may run concurrently. For the mapping policy, two policies – round-robin and back-to-back – were described, but only the former was evaluated. Likewise, for the running policy, two policies – one-at-a-time and all-at-once – were described, but in this case both were evaluated. By combining these policies with different organizations that supported up to a total of 16 tasks and 16 processing units, this work considered multiscalar processor designs along a continuum of cost-performance points.

With respect to one-to-one task assignment, performance improvements were linear as the number of tasks in concurrent execution (and hence the number of processing units) were doubled. Using from 2 to 16 tasks, the harmonic mean of the instructions per cycle ranged from 1.50 to 3.50 for the SPEC CINT95 benchmarks and from 2.00 to 6.00 for the SPEC CFP95 benchmarks. Nonetheless, since these performance improvements seemed modest in relation to cost necessary to achieve them, with a decrease in resource utilization for every increase in processing capacity, many-to-one task assignment was explored as an alternative. With respect to many-to-one task assignment, the performance of the most promising configuration (of the many considered), 4 processing units running 4 tasks all-at-once per processing unit (a total of 16 tasks), was able to nearly match (90%) the performance of 16 processing units running 1 task per processing unit (which was the highest in this study), but at a fraction of the cost (25%). To be specific, the harmonic mean of the instructions per cycle reached 3.00 (as compared to 3.50) for the SPEC CINT95 programs and 5.50 (as compared to 6.00). for the SPEC CFP95 programs.

8.2.2 Intra-Task Aspect

In studying the intra-task aspect of processing unit design, the following three microarchitectural features of the individual processing units were identified as the key design choices: the instruction window, the instruction selection and issue, and the instruction execution characteristics.

First, the effects of different in-flight and on-deck configurations were described and evaluated for the instruction window. In the case of multiscalar processors, it was found that the instruction window of each processing unit does not affect performance to as great an extent as in conventional superscalar processors. The results showed that a multiscalar processor using processing units (each) with an instruction window of only 16 instructions in-flight (decoded) and 8 instructions on-deck (decoded, but not issued), achieved comparable performance to an instruction window of 64 instructions in-flight and 64 instructions on-deck.

Second, the impact of multiple instruction issue and dynamic instruction scheduling were described and evaluated for the instruction selection and issue. For increases in issue width, it was found that performance grows steadily, with the knee at 2-wide, for either in-order or out-of-order designs. Moreover, this growth in performance was observed for multiscalar processors with few as well as many processing units, but was more significant for more processing units. However, it was also observed that the out-of-order growth rate was higher than the in-order growth rate, yielding a performance crossover point (between 2-wide out-of-order and 4-wide in-order) where a lower issue width out-of-order processing unit outperformed a higher issue width in-order processing unit.

Third, the effects of the number and type of functional units on execution were described and evaluated for instruction execution. In order to do so, the types of functional units were classified as compute (integer and floating point), memory, and finally control, with their number varied for out-of-order issue widths of 1, 2, and 4 instructions per cycle. After a variety of configurations were explored, for both the SPEC CINT95 and the SPEC CFP95 benchmarks, it was concluded, for issue widths beyond 1-wide, that 1 control functional unit was always sufficient, and at least 2 compute and 2 memory functional units were necessary to provide performance comparable to the use of uniform functional units.

8.3 Instruction Supply

The combined behavior of the control flow predictor and the instruction memory determines the characteristics of program sequencing, and therefore the performance effect that the instruction supply may have on a multiscalar processor. For this reason, the study of the instruction supply for a multiscalar processor performed in this work focused on the design and evaluation of the components of the hierarchical predictor and the instruction memory.

8.3.1 Hierarchical Predictor

In terms of the hierarchical predictor, this thesis has shown that a straightforward approach using components based on existing prediction technology can provide for both the inter-task and intra-task prediction aspects of a multiscalar processor. The experiments performed as a part of this work indicated that both inter-task prediction and intra-task prediction can be important performance factors, but that it is usually the case that overall performance is more sensitive to the inter-task than to the intra-task prediction accuracy. Moreover, this work showed that varying the amount of prediction state from 8K-entry to 64K-entry for each individual predictor (inter-task and intra-task) provided no appreciable difference in prediction accuracy among these sizes.

The inter-task and intra-task predictors were evaluated in isolation of one another using several well-known prediction schemes (address-based, path-based, global-pattern-based, and self-pattern-based) for both predictors. Among these predictors, it was observed that the path-based and global-pattern-based schemes consistently performed best, for both inter-task and intra-task prediction, across the range of SPEC CINT95 and SPEC CFP95 programs. Using these results factored in with design complexity, it was concluded that the best hierarchical prediction scheme may be provided by combining a path-based inter-task predictor with a global-pattern-based intra-task predictor. Still, comparisons with perfect predictors indicated that there is considerable performance improvement possible with respect to the SPEC CINT95 benchmarks (more for inter-task than for intra-task), but little or none with respect to the SPEC CFP95 benchmarks (for either inter-task or intra-task).

8.3.2 Instruction Memory

This thesis has shown that, just as for the hierarchical predictor, components based on existing cache technology can be used to provide the inter-task and intra-task instruction memory aspects.

A task cache was used to handle the inter-task aspect, and an instruction cache was used to handle the intra-task aspect. Both caches were studied according to typical cache parameters: miss rate, effective access latency, and overall impact on performance for a variety of sizes and associativities. Yet, because the instruction cache involved additional considerations of high-bandwidth and low-latency access, it was evaluated using two suitable well-known organizations, interleaved shared cache and duplicated private cache, as well as using a hybrid organization of both private and shared caches.

The experiments which focused on miss rate demonstrated that the task and instruction caches of a multiscalar processor behave in a typical cache-like manner. With respect to both the inter-task and intra-task aspects of the instruction memory, miss rates decreased steadily as the cache size increased (until the working set fit into the cache) and as the associativity was increased (with the improvement in going from 1-way to 2-way more significant than going form 2-way to 4-way).

The experiments which focused on effective access latency demonstrated, as one might expect, that overall performance decreased as latency increased, either due to an increase in access time to service misses or due to an increase in access time to service hits. With regard to inter-task and intra-task instruction memory aspects, this behavior was far more pronounced in the task cache than it was in the instruction caches. In fact, the results indicated that though the instruction caches exhibited some degree of latency tolerance in terms of performance for high effective access latencies, the task cache exhibited none.

Even so, the demands, in terms of high-bandwidth and low-latency access, placed upon the instruction cache made its design more challenging than that of the task cache. Of particular significance, it was determined that private and shared instruction cache organizations were only able to provide comparable performance when the individual storage of each private cache was comparable to the overall storage of the shared instruction cache. Furthermore, though the hybrid instruction cache was able to mitigate these difficulties somewhat, its success in doing so decreased as the number of processing units increased (with the break-even point around 4 processing units).

8.4 Data Supply

The study of the data supply for a multiscalar processor performed in this thesis focused on the design and evaluation of the components of the register file and data memory. These components were chosen for study because the combined behavior of the register file and the data memory determines the characteristics of communication between producers and consumers of data values, and thereby the performance impact that the data supply may have on a multiscalar processor.

8.4.1 Register File

The work involving the design and evaluation of the register file performed in this thesis has demonstrated that a decentralized organization, in which the complexity was placed in software rather than in hardware, can support the speculative register access needs of a multiscalar processor, with both high-bandwidth and low-latency. In this work, the register file design was considered with respect to both its behavior and its impact on performance.

From the study of register file behavior, it was concluded that sufficient register communication locality exists among tasks to support an effective and efficient decentralized register file design. First, it was found that the filtering that naturally occurred as new register values were created and old register values were killed held register traffic in check. Second, it was determined that the register task travel distance was also held in check, due to the same filtering effect. Third, it was observed that though the total register traffic per cycle increased steadily as the number of tasks in concurrent execution increased, the link register traffic per cycle actually decreased as the number of tasks increased.

From the study of the performance impact of the register file, it was determined that the register file latency rather than the register file bandwidth was the critical factor. In particular, it was observed that the processor performance improved markedly and at a fairly high rate as the register file latency was decreased. However, though it was observed that the processor performance improved as the register file bandwidth increased, it did so only marginally and at a fairly low rate. Moreover, it was concluded that register file latency (to a large extent) and bandwidth (to a small extent) have a dampening effect that reduces the performance potential of multiscalar processors.

8.4.2 Data Memory

The work involving the design and evaluation of the data memory performed in this thesis has demonstrated that a decentralized organization based solely on hardware can support the high-bandwidth and low-latency, speculative memory access demands of a multiscalar processor. This decentralized organization comprised a more or less conventional data cache and two specialized components, the address resolution buffer and the memory dependence table. In studying these components, it was found that, though the data cache determined the level of performance the data memory was able to attain, the address resolution buffer and the memory dependence table determined the amount of performance the data memory was able to sustain.

Considering the attainable level of performance, the data cache experiments indicated that a reasonable sized data cache (64K-byte) with a moderate amount of associativity (2-way) can provide nearly the highest performance possible on the SPEC CINT95 and SPEC CFP95 benchmarks for the processor configurations involved in this study. Moreover, considering the sustainable amount of performance, this study indicated that both the address resolution buffer and the memory dependence table were critical performance factors that if not handled properly reduced the overall performance level significantly. However, the results of this work identified that though the address resolution buffer and the memory dependence table manifested the same effect, reduced performance, each did so in a different manner.

The address resolution buffer was found to reduce performance as a result of processor stall cycles that occurred when it overflowed and was no longer able to track additional loads and stores. The studies of the address resolution buffer performed in this thesis identified the causes of such overflow as insufficient number of entries, insufficient associativity among the entries, and/or insufficient version storage per entry. Furthermore, of these three causes, it was concluded that the amount of version storage per entry was the actual root cause for most cases of address resolution buffer overflows. Taken altogether, the results indicated that the use of a version that was consistent with the data cache block size was the proper choice since it allowed the address resolution buffer to better capture the same spatial and temporal locality as the data cache itself.

The memory dependence table was found to reduce performance as a result of processor squash (rather than stall) cycles when it was unable to synchronize the blind speculative execution of loads and stores. The experiments conducted in this focused on the performance impact with or without the use of the memory dependence table, instead of the specifics of its organization, to reach this conclusion. Nevertheless, this work made the important observation that the role of the memory dependence table increased in significance as the size of the task window on the dynamic instruction sequence increased. To be specific, there was no appreciable difference in performance with or without the memory dependence table for small, less aggressive, multiscalar processor configurations (such as for 2 processing units or 2 tasks); there was, however, a significant performance difference for large, more aggressive, multiscalar processor configurations (such as for 16 processing units or 16 tasks).

8.5 Performance Comparison

To complement the design and evaluation of a multiscalar processor, additional results were gathered to provided a performance comparison, relative to a baseline 1-wide out-of-order issue processor, between realistic multiscalar processors and idealistic superscalar processors. Though, the multiscalar processors and superscalar processors were comparable in terms of their instruction and data memory systems as well as their overall processing capabilities, the instruction windows from which instruction-level parallelism was extracted were quite different because of the fundamental differences in the two paradigms. Furthermore, the multiscalar processors used in the performance comparison were evaluated with two different data latency configurations, one conservative and the other aggressive.

Overall, the multiscalar configurations that compared most favorably to the superscalar configurations across both the SPEC CINT95 and SPEC CFP95 benchmarks were those with the highest number of tasks in concurrent execution. In particular, with respect to the microarchitecture and compiler capabilities assumed in this thesis, the collection of results indicated that multiscalar processors outperformed superscalar processors over nearly the entire range of configurations for the SPEC CFP95 programs, but over only a selected range of configurations for the SPEC CINT95 programs. From a thoughtful examination of these ranges, it was concluded that the ability of a multiscalar processor to provide better performance than a superscalar processor was influenced primarily by the number (and characteristics) of the tasks in concurrent execution on the processor and secondarily by the overall issue width of the processor.

8.6 Future Directions

Though this work has tried to be comprehensive, the actual design space for multiscalar processors is enormous. As a result, there remain many questions that need to be answered concerning multiscalar processors. A limitation of this work which needs to be addressed is that a deeper understanding is required for each of components studied in this thesis. This work has analyzed the performance trade-offs to some extent, but what is needed is a fine-tuning of this analysis. Each component needs a detailed and thorough analysis to identify the cost-performance trade-offs that guide the characteristics of its actual implementation.

Though this level of detail was beyond the scope of this thesis (with so many components studied), it is nevertheless a crucial factor for the assessment of the complexity and performance of an actual implementation of a multiscalar processor. More importantly, this level of detail is particularly important since it is needed to provide an estimate of clock speed, the key performance factor that could not be accounted for in this work. With such details clarified, the results presented in this work may then be reconciled to ascertain where multiscalar processors fall in the performance spectrum.

Bibliography

[1] R. D. Acosta, J. Kjelstrup, and H. C. Torng. An instruction issuing approach to enhancing performance in multiple functional unit processors. *IEEE Transactions on Computers*, 35(9):815–828, September 1986.

[2] T. Agerwala and J. Cocke. High performance reduced instruction set processors. Technical report, IBM Systems Journal, March 1987.

[3] Saman P. Amarasinghe, Jennifer M. Anderson, Chris S. Wilson, Shih-Wei Liao, Brian R. Murphy, Robert S. French, Monica S. Lam, and Mary W. Hall. Multiprocessors from a software perspective. *IEEE Micro*, 16(3), June 1996.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings AFIPS Spring Joint Computer Conference*, Atlantic City, New Jersey, April 1967.

[5] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H.T. Kung, Monica Lam, Onat Menzilcioglu, and Jon A. Webb. The warp computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, 36(12):1523–1538, December 1987.

[6] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[7] Semiconductor Industry Association. The national technology roadmap for semiconductors, 1995.

[8] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 20(2), May 1992.

[9] C. Gordon Bell. Multis: A new class of multiprocessor computers. *Science*, pages 462–467, April 1985.

[10] Dileep Bhandarkar and Jason Ding. Performance characterization of the Pentium Pro processor. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 288–297, San Antonio, Texas, February 1–5, 1997. IEEE Computer Society TCCA.

[11] Mark T. Bohr. Interconnect scaling - the real limiter to high performance ULSI. *Solid State Technology*, 39(9), September 1996.

[12] Brian K. Bray and M. J. Flynn. Strategies for branch target buffers. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 42–50, Albuquerque, New Mexico, November 18–20, 1991. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[13] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow. Single instruction stream parallelism is greater than two. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 276–286, Toronto, Ontario, May 27–30, 1991. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 19(3), May 1991.

[14] Brad Calder and Dirk Grunwald. Fast & accurate instruction fetch and branch prediction. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 2–11, Chicago, Illinois, April 18–21, 1994. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 22(2), April 1994.

[15] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David P. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, Palo Alto, California, October 5–8, 1987. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 15(5), October 1987; *Operating Systems Review,* 21(4), October 1987; *SIGPLAN Notices,* 22(10), October 1987.

[16] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–344, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 23(2), May 1994.

[17] David E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–150, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 16(2), May 1988.

[18] K. Diefendorff and M. Allen. Organization of the motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(2), April 1992.

[19] Simonjit Dutta and Manoj Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 258–263, Ann Arbor, Michigan, November 29–December 1, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[20] Susan J. Eggers. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessors*. PhD thesis, University of California, Berkeley, 1989.

[21] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 149–159, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[22] Keith I. Farkas and Norman P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, Chicago, Illinois, April 18–21, 1994. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 22(2), April 1994.

[23] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 78–89, Raleigh, North Carolina, January 22–25, 1995. IEEE Computer Society TCCA.

[24] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, Stockholm, Sweden, June 13–17, 1983. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 11(3), June 1983.

[25] Manoj Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin - Madison, 1993.

[26] Manoj Franklin. Multi-version caches for multiscalar processors. In *Proceedings of the 1st International Conference on High Performance Computing*, New Delhi, India, December 1995.

[27] Manoj Franklin and Guri S. Sohi. ARB: A hardware mechanism for the dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[28] Manoj Franklin and Gurindar S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–245, Portland, Oregon, December 1–4, 1992. IEEE Computer Society TC-MICRO and ACM SIGMICRO. SIG MICRO Newsletter 23(1–2), December 1992.

[29] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 16(2), May 1988.

[30] Sridhar Gopal, T.N. Vijaykumar, James E. Smith, and Guri S. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 195–205, Las Vegas, Nevada, February 1–4, 1998. IEEE Computer Society TCCA.

[31] G. F. Grohoski. Machine organization of the IBM RISC system/6000 processor. *IBM Journal of Research and Development*, 34:37–58, January 1990.

[32] L. Gwennap. Digital leads the pack with 21164. *Microprocessor Report*, 8(12), September 12 1994.

[33] L. Gwennap. MIPS r10000 uses decoupled architecture. *Microprocessor Report*, 8(14), October 24 1994.

[34] L. Gwennap. PA-8000 combines complexity and speed. *Microprocessor Report*, 8(15), November 14 1994.

[35] L. Gwennap. PPC 604 powers past pentium. *Microprocessor Report*, 8(5), April 18 1994.

[36] L. Gwennap. Ultrasparc unleashes SPARC performance. *Microprocessor Report*, 8(13), October 3 1994.

[37] Quinn Jacobson, Steve Bennett, Nikhil Sharma, and James E. Smith. Control flow speculation in multiscalar processors. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 218–229, San Antonio, Texas, February 1–5, 1997. IEEE Computer Society TCCA.

[38] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1990.

[39] R. Jolly. A 9-ns 1.4 gigabyte/s, 17-ported CMOS register file. *IEEE Journal of Solid-State Circuits*, 25:1407–1412, October 1991.

[40] Norman P. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers*, 38(12):1645–1658, December 1989.

[41] Norman P. Jouppi and Steven J. E. Wilton. Tradeoffs in two-level on-chip caching. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 34–45, Chicago, Illinois, April 18–21, 1994. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 22(2), April 1994.

[42] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 34–42, Toronto, Ontario, May 27–30, 1991. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 19(3), May 1991.

[43] Stephen W. Keckler and William J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202–213, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 20(2), May 1992.

[44] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87. IEEE Computer Society and ACM SIGARCH, May 1981.

[45] S. Laha, J. Patel, and R. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, November 1988.

[46] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 20(2), May 1992.

[47] Alvin R. Lebeck and Guri S. Sohi. Request combining in multiprocessors with arbitrary interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, November 1994.

[48] J. K. F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, January 1984.

[49] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 226–237, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[50] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Cambridge, Massachusetts, October 1–5, 1996. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 24, October 1996; *Operating Systems Review,* 30(5), December 1996; *SIGPLAN Notices,* 31(9), September 1996.

[51] J. S. Liptay. Structural aspects of the system/360 model 85 part ii: the cache. Technical report, IBM Systems Journal, July 1968.

[52] E. P. Markatos and T. J. LeBlanc. Load balancing versus locality management in shared-memory multiprocessors. In *Proceedings of the 1992 International Conference on Parallel Processing*, pages 258–267, August 1992.

[53] Scott McFarling. Combining branch predictors. Technical Report WRL-TN-36, WRL Technical Note, June 1993.

[54] Scott McFarling and John Hennessy. Reducing the cost of branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, Tokyo, Japan, June 2–5, 1986. IEEE Computer Society TCCA, ACM SIGARCH, and the Information Processing Society of Japan. *Computer Architecture News,* 14(2), June 1986.

[55] Wen mei Hwu and Yale N. Patt. HPSm, a high performance restricted data flow architecture having minimal functionality. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 297–306, Tokyo, Japan, June 2–5, 1986. IEEE Computer Society TCCA, ACM SIGARCH, and the Information Processing Society of Japan. *Computer Architecture News,* 14(2), June 1986.

[56] Stephen Melvin and Yale Patt. Exploiting fine-grained parallelism through a combination of hardware and software techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 287–296, Toronto, Ontario, May 27–30, 1991. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 19(3), May 1991.

[57] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, pages 181–193, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society TCCA.

[58] Kazuaki Murakami, Naohiko Irie, Morihiro Kuga, and Shinji Tomita. SIMP (Single Instruction stream/Multiple instruction Pipelining): A novel high-speed single-processor architecture. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 78–85, Jerusalem, Israel, May 28–June 1, 1989. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 17(3), June 1989.

[59] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 15–23, Ann Arbor, Michigan, November 29–December 1, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[60] Alexandru Nicolau and Joseph A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, 33(11):968–976, November 1984.

[61] R. R. Oehler and R. D. Groves. IBM RISC system/6000 processor architecture. *IBM Journal of Research and Development*, 34:23–36, January 1990.

[62] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh*

International Conference on Architectural Support for Programming Languages and Operating Systems, pages 2–11, Cambridge, Massachusetts, October 1–5, 1996. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 24, October 1996; *Operating Systems Review,* 30(5), December 1996; *SIGPLAN Notices,* 31(9), September 1996.

[63] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford Univerity, Computer Systems Lab, February 1997.

[64] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Quantifying the complexity of superscalar processors. Technical Report 1328, University of Wisconsin - Madison, December 1996.

[65] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, pages 206–218, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society TCCA.

[66] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Massachusetts, October 12–15, 1992. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 20, October 1992; *Operating Systems Review,* 26, October 1992; *SIGPLAN Notices,* 27(9), September 1992.

[67] Y. N. Patt, S. W. Melvin, W. W. Hwu, and M. Shebanow. Critical issues regarding HPS, a high performance microarchitecture. In *Proceedings of the 18th Annual Workshop on Microprogramming*, pages 109–116, December 1985.

[68] Yale N. Patt, Wen mei Hwu, and Michael Shebanow. HPS, a new microarchitecture: Rationale and introduction. In *Proceedings of the 18th Annual Workshop on Microprogramming*, pages 103–108, December 1985.

[69] A. R. Pleszkun and G. S. Sohi. The performance potential of multiple functional unit processors. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 37–44, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 16(2), May 1988.

[70] Dionisios N. Pnevmatikatos, Manoj Franklin, and Gurindar S. Sohi. Control flow prediction for dynamic ILP processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 153–163, Austin, Texas, December 1–3, 1993. IEEE Computer Society TC-MICRO and ACM SIGMICRO. SIG MICRO Newsletter 24, December 1993.

[71] B. R. Rau, D. W. L. Yen, W. Yen, and R. Towle. The cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *IEEE Computer*, 22(1):12–35, January 1989.

[72] B. Ramakrishna Rau. Pseudo-randomly interleaved memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 74–83, Toronto, Ontario, May 27–30, 1991. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 19(3), May 1991.

[73] Jeff Reilly. A brief introduction to the spec cpu95 benchmark. *IEEE-CS TCCA Newsletter*, June 1996.

[74] Matt Reilly and John Edmondson. Performance simulation of an alpha microprocessor. *IEEE Computer*, 31(5):50–58, May 1998.

[75] MicroDesign Resources. Special issue: Celebrating the 25th anniversary of the microprocessor. *Microprocessor Report*, 10(10), August 5 1996.

[76] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[77] M. Slater. AMD's k5 designed to outrun pentium. *Microprocessor Report*, 8(14), October 24 1994.

[78] Alan J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[79] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon. The ZS-1 central processor. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–204, Palo Alto, California, October 5–8, 1987. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 15(5), October 1987; *Operating Systems Review,* 21(4), October 1987; *SIGPLAN Notices,* 22(10), October 1987.

[80] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–145, May 1981.

[81] James E. Smith. Decoupled access/execute architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, pages 112–119, April 1982.

[82] James E. Smith. Dynamic instruction scheduling and astronautics ZS-1. *IEEE Computer*, 22(7):21–35, July 1989.

[83] James E. Smith and Gurindar S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, December 1995.

[84] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 18(2), June 1990.

[85] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, pages 194–205, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society TCCA.

[86] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptable, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[87] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 23(2), May 1994.

[88] Gurindar S. Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review,* 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[89] Gurindar S. Sohi and Sriram Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34, Pittsburgh, Pennsylvania, June 2–5, 1987. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 15(2), June 1987.

[90] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 2–13, Las Vegas, Nevada, February 1–4, 1998. IEEE Computer Society TCCA.

[91] J. E. Thornton. Parallel operation in the control data 6600. In *Proceedings AFIPS Spring Joint Computer Conference*, 1964.

[92] J. E. Thornton. *Design of a Computer – The Control Data 6600.* Scott, Foresman, and Company, 1970.

[93] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1990.

[94] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 23(2), May 1994.

[95] Augustus K. Uht. *Hardware Extraction of Low-level Concurrency from Serial Instruction Streams*. PhD thesis, Carnegie-Mellon University, 1985.

[96] T. N. Vijaykumar, Scott E. Breach, and Gurindar S. Sohi. Register communication strategies for the multiscalar architecture. Technical Report 1333, University of Wisconsin - Madison, March 1997.

[97] T.N. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, University of Wisconsin - Madison, 1998.

[98] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review,* 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[99] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison–Wesley Publishing Company, Inc., 1988.

[100] Kenneth M. Wilson and Kunle Olukotun. Designing high bandwidth on-chip caches. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, pages 121–132, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society TCCA.

[101] Andrew Wolfe and John P. Shen. A variable instruction stream extension to the VLIW architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review,* 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[102] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Conference Proceedings, 1993 International Conference on Supercomputing*, pages 67–76, Tokyo, Japan, July 20–22, 1993. ACM SIGARCH.

[103] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, Albuquerque, New Mexico, November 18–20, 1991. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[104] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 20(2), May 1992.

[105] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 129–139, Portland, Oregon, December 1–4, 1992. IEEE Computer Society TC-MICRO and ACM SIGMICRO. SIG MICRO Newsletter 23(1–2), December 1992.

[106] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, San Diego, California, May 17–19, 1993. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 21(2), May 1993.

[107] Albert Yu. The future of microprocessors. *IEEE Micro*, 16(6), December 1996.