Optimizing Inter-Instruction Value Communication through Degree of Use Prediction

by

Jeffrey Adam Butts

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
University of Wisconsin–Madison
2004

Abstract

The design of high-performance value communication structures represents a significant challenge in the implementation of future microprocessors. The complexity of these structures—namely, the register file, the bypass network, and the instruction window—dwarfs that of the actual instruction execution hardware and has become the predominant factor in determining the maximum operating frequency. Being wire-dominated, these structures also benefit less from technology scaling than the execution circuitry, increasing the importance of this problem.

Value communication structures are complex because they are overly general. Each of these structures is built to support the most general possible communication pattern for each value. However, examining value communication in actual programs reveals that most values are involved in simple communication, such as producer to single consumer. This dissertation proposes an alternate model for inter-instruction register value communication in which values may be handled differently by mechanisms optimized to their individual needs.

The foundation for this model is knowledge of communication on a per-value basis. The communication resource requirements of a value are summarized by its degree of use, which is simply the number of consumers of that value. Accurate degree of use information may be obtained speculatively via a prediction mechanism. This information may then be used to handle each value in the most efficient manner available.

This dissertation makes three major contributions to the development of this communication model. First, an in-depth study of register value communication in terms of the degree of use is presented in order to demonstrate the opportunity for alternative communication mechanisms. Second, methods for the accurate prediction of degree of use are presented and characterized. Finally, two proof-of-concept applications of the degree of use knowledge are developed. Useless instruction elimination exploits the occurrence of degree of use zero values, which have no consumers. Avoiding the execution of instructions generating such values reduces resource utilization, and, under resource contention, improves performance. Use-based register caching applies degree of use information to the management of a register cache. The resulting insight into each value's communication behavior allows the limited cache space to be used more effectively than previously proposed register caches.

Acknowledgements

Above all others, I want to thank my wife, Susan Kalis. A Ph.D. candidate herself, she has been remarkably understanding of the afflictions associated with being a grad student: strange hours, long binges at the computer, lack of motivation, inability to maintain a social life, and general malaise. Especially during the past several months, she has put her own needs aside in order to support me during my final push to graduate. I am looking forward to returning the favor. It is said that it is easier to suffer together than to suffer alone—truly, I would endure any hardship so long as she was there with me. I would not be here, nor would I want to, without her, so this work is dedicated to her.

I owe a tremendous amount to my parents, Jeffrey and Stephanie. Rooted in the pacific northwest, they have seen little of me since I left to attend college in the midwest 11 years ago. However, knowing that I had a home always open to me has been a source of strength. It is their upbringing to which I credit being one of the most emotionally stable people I know. They have always encouraged me to excel and provided an environment in which I could do so. I can only aspire to someday be as good a parent as each of them was to me.

My advisor Guri Sohi has been instrumental in making me the researcher that I am today. He spent the better part of the last six years trying to break me of my engineering mentality, encouraging me to step back and look at the bigger picture instead of getting bogged down in gates and propagation delays. He has taught me a great deal not only about the field of computer architecture, but about the practice of performing good research. I am also grateful for his inhuman patience with my . . . flexible approach to time management.

In a very real sense, I have had many research advisors. An incredible strength of the computer science department at the University of Wisconsin is the engagement of the faculty in the development of each and every graduate student. I have benefitted from discussions with Ras Bodik, Charlie Fischer, Jim Goodman, Mark Hill, Susan Horwitz, and Mary Vernon. David Wood in particular has been a tremendous resource in matters both technical and otherwise. I am also happy to count him among my friends (and fellow left wings).

All of the members of my thesis committee deserve credit for the research contained herein (although I bear full responsibility for its shortcomings). In addition to Guri and David, Charlie

Fischer, Mikko Lipasti, and Jim Smith all contributed to my development as an architect. Aside from their direct input on my research both past and present, I have learned from their own research. Perhaps most of all, though, I have benefited from the opportunity to observe them on others' committees and at conferences, where I was exposed to their thought processes and analyses of different research ideas.

For reasons still opaque to me, Amir Roth, Craig Zilles, and Milo Martin befriended me back when I was a know-it-all first-year student, despite being several years my senior. They are three of the smartest people I have ever met, and it was an invaluable experience to watch them progress through and succeed in graduate school. I have tried hard to emulate them, which has probably been a significant factor in my own graduation.

Many other current and former students and staff in the computer science department at Wisconsin have also been friends and collaborators including Ross Dickson, Brian Fields, Allison Holloway, Carl Mauer, David Parter, Erik Paulson, and Ravi Rajwar. David Parter has my extra thanks for opening his home to me as my Madison base since I relocated to Chicago last year.

My graduate school experience would have been less productive and much less fun were it not for the great company of so many other graduate students. Of these, Brandon Schwartz and Paramjit Oberoi deserve special mention. Brandon was my roommate for most of graduate school, and, in addition to sitting and discussing microarchitecture in front of the 4-by-8 foot dry erase board in our apartment, we engaged in endless post-midnight discourses on computer science, physics, philosophy, psychology, sociology, political science, and hockey among other topics. Param and I had many equally wide-ranging discussions, especially after he became my officemate. Param also endured having me as a partner for not one class project, but every project in all four classes we took together. Both Brandon and Param helped me refine nearly every idea in this work and many more that are not.

A conservative estimate puts the total computer time spent obtaining the data in this document at over 100,000 hours (almost 11.4 *years*), although it was generated in only a few months of calendar time. The Condor system made this possible, and I am grateful to the members of the Condor team for providing and supporting this service. They often went the extra mile to scrounge additional machines to help me meet various deadlines.

I am also extremely thankful for all of the people, those I know personally and otherwise, who maintain the amazing research support infrastructure in the computer science department. The staff of the computer systems lab performs the monumental—and often thankless—task of keeping the computers and the network up and running. It is a testament to their success that I find it difficult to express the extent of their role: everything just works, allowing people to concentrate fully on their research.

I still find it remarkable that I was able to get paid to learn for the past six years. For this I am deeply indebted to the Fannie and John Hertz Foundation and Intel Corporation. In addition to its generous financial support over five years, the Hertz Foundation has also been a valuable source of contacts and an advocate for my personal career. For the final year of my graduate career, it was a fellowship from Intel that allowed me to maintain focus on my research instead of the source of my next meal.

Finally, I would be remiss if I did not at least mention several others who have helped me reach this point. Kathy Pfaendler, George Delegans, Nels Doeleman, Richard Green, Ken Houle, Carl Simonsen, Youssef El-Mansy, Greg Taylor, and Jeff Smith have all had profound effects on my life and intellectual development.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

At a very high level, the operation of a general-purpose computer can be broken into two tasks: the manipulation of data by individual instructions and the communication of data values among the appropriate instruction instantiations. Of these two tasks, inter-instruction value communication is vastly more difficult and is the origin of much of the complexity found in high-performance processors today.

This dissertation explores the relationship of a value's degree of use to the nature of its communication. A value's degree of use is simply the number of times the value is used as an input by the successive instructions of the program, and it is an indicator of the resources needed to distribute that value to its consumers. Most values exhibit very simple communication patterns, yet current implementations handle all values uniformly. The resulting communication inefficiency manifests as inflated complexity, communication latency, and power dissipation.

The application of speculative degree of use information allows for the handling of each value to be tuned to its particular characteristics. Two such optimizations are presented: one completely avoids the creation of a value when it will not be communicated to any subsequent instructions, while the other applies degree of use knowledge to manage the set of values kept in a small, low-latency register cache.

1.1 Inter-Instruction Value Communication

Inter-instruction value communication is a need common to all general-purpose processing architectures. All such architectures provide primitives—instructions—for the manipulation of certain data representations. The operation of a computer entails the execution of instructions on a set of data according to a program. The actions specified by the individual instructions are generally simple and have changed little since the introduction of the first programmable computers. The versatility of a computer results from the ability to describe arbitrarily-complicated operations in terms of these simple instructions. Regardless of the variety and richness of the available instructions, however, nearly all tasks of interest will require the sequential application of multiple primitives. Thus, *the communication of data values between the instructions constituting a program is a fundamental aspect of computing*.

The primary motivation for this work is that this task of inter-instruction value communication is difficult. While part of this difficulty is inherent to the actual communication of values, much of it results from how communication mechanisms are implemented in current architectures. The sequential, register-based architecture is the basis for the majority of modern general-purpose processor implementations. Most value communication in this architecture occurs through a limited number of storage locations called registers, and it is this class of communication that is the focus of this dissertation. Achieving high performance under this model requires complex value communication mechanisms. Technology trends will render current methods of register value communication inadequate for future high-performance implementations.

1.1.1 Inter-instruction communication in sequential architectures

In a sequential architecture, inter-instruction communication is specified in a program via named storage locations. Each instruction *addresses* (i.e., specifies) storage locations where its inputs may be found and where its output should be placed. Thus, value communication is specified indirectly: instructions do not name the consumers of their result, nor do the consumers name the producers of their inputs. Instead, a value may be communicated between two instructions if they name a common storage location. This condition is necessary but not sufficient. The two instructions must occur in the proper temporal sequence, with the producer of the value occurring prior to the consumer. Additionally, no intervening instruction can place its result in the storage loca-

tion or it would become the producer. The sequence of instructions within the program, descriptively called *program order*, is therefore fundamental to the specification of value communication, and this is what gives the sequential processing model its name.

It is important here, and throughout the remainder of this document, to distinguish between two different types of instructions. A *static instruction* is an atom of a program; it specifies a single kind of supported operation (e.g., addition), and the locations of its input(s) and output(s). As its name implies, it does not change during the execution of the program.† A given static instruction may execute many times on different values by virtue of the storage location(s) it specifies containing different values each time. One such instance of a static instruction is called a *dynamic instruction*. A dynamic instruction only occurs once; its inputs are fixed to the contents of the named storage locations at the time of its instantiation.

1.1.2 Register value communication

Two different classes of storage locations may be named by the static instructions of the program: registers and memory. Registers comprise a small number—architecture dependent, but on the order of a few tens—of storage locations that are named directly. In other words, given a static instruction that names a particular register as an input, *every* dynamic instance of that instruction will also receive its input from the same register. The capacity of memory is vastly larger (ideally infinite). The number of individually-addressable memory locations hinders the direct addressing of particular memory location (although some architectures support it); generally, a memory location is named with the aid of a value stored in a register.

This dissertation focuses on the value communication occurring through registers because it is the dominant mode of inter-instruction communication, in spite of the much smaller register namespace. Register-based architectures are aptly named: nearly every useful instruction specifies at least one register as a source or destination. *Load-store architectures* are a subclass of register-based architectures in which only two specific classes of instructions can address memory. A *load* moves a value from memory to a register, while a *store* performs the reverse operation. In each case, another register contains a value used in addressing the memory location. While all

† Stored-program computers do not distinguish between instructions and other kinds of program data. A few architectures allow running programs to modify their program code; the occurrence of such *self-modifying code* is rare even where supported.

other types of operations may communicate only via registers, these memory operations still use two registers each, illustrating the importance of register value communication. Architectures such as the IA-32 allow instructions to specify a memory location as a source or destination; however, most implementations convert these instructions into a sequence of simpler ones communicating via registers and using loads or stores as needed [44].

1.1.3 Meeting the demands of parallelism

Achieving high-performance in a sequential architecture necessitates overcoming an inherent disadvantage of the programming model. While a sequential program imposes a total ordering among all instructions, a given instruction frequently does not depend on the execution of all prior instructions. The useful consequence of this fact is that the execution of instructions that are independent of one another can take place simultaneously, reducing the total time required to execute all of the operations specified by the program. The existence of independent instructions within a program is referred to as *instruction-level parallelism*.

Exploiting parallelism for high-performance using register-based communication demands: (1) many register storage locations, (2) high register access bandwidth, (3) many communication endpoints, and (4) low access latency. Each simultaneously-executing instruction requires input values from storage (some of which may be shared) and storage for its result, requiring more total register storage than if execution occurred one-instruction-at-a-time. Even if few unique register names are available, register renaming enables more physical storage locations to be in use concurrently than the number of register names would otherwise support.† The adoption of simultaneous multi-threading [85], which increases available parallelism by offering multiple execution contexts with their own register namespaces, requires even more register storage [13]. While the quantity of register storage grows with the degree of parallelism, so too does the access bandwidth required of that storage. Each instruction must access the register storage to retrieve its input operands and store its result value; simultaneously-executing instructions must perform these

† Where parallelism exists, it can be obscured by false dependences introduced as a result of a limited register namespace. Sequences of instructions belonging to otherwise independent computations are serialized merely by virtue of specifying the same register at some point. Register renaming maps the limited number of names for register locations (the architectural registers) to a much larger—and implementation dependent—number of storage locations for actual values (the physical registers). Physical register identifiers are substituted for architectural register names such that real data-dependences are preserved.

accesses concurrently. Furthermore, the instructions will occupy different execution resources, each of which needs its own connections to the register storage. Finally, the latency of the access to register values must not suffer unduly: the benefit of executing many instructions at a time is diminished if the instructions take much longer to obtain their input values.

Much of the complex circuitry in current processors exists to support these demands of register value communication. Besides the register file itself, this circuitry includes the bypass network and the instruction scheduling hardware (in dynamically-scheduled processors, which are the focus of this work) [66]. The bypass network provides direct interconnection among different execution units so that latency-critical value communication may occur without traversing the register storage. The instruction scheduler enforces program data-dependences by allowing instructions to execute only after their input values are available; thus, its design is heavily impacted by the nature of inter-instruction communication. In some instances, the instruction scheduling apparatus even stores register values [11, 67, 83]. The resources spent supporting inter-instruction value communication dwarf those spent to actually execute instructions. Supporting high-performance value communication with these structures represents a significant challenge facing future designs.

1.1.4 The problem

Value communication mechanisms benefit less from technology scaling than other types of circuitry because they are large, centralized, and wire-dominated [66]. First, the quantity of storage accessible in a fixed period of time relative to the latency of a fixed computational operation (e.g., addition) is decreasing [1]. Thus, the relative latency of the access to a fixed amount of register storage will increase. If the amount of register storage must also increase, the effect will be correspondingly larger. Supporting more simultaneous accesses to the register storage has a penalty similar to that of increasing its capacity. A second—and not unrelated—trend is the smaller relative improvement in the speed of wires versus transistors as semiconductor technology moves forward [10].† Slow interconnection imposes a cost for increasing the number of execution units that must access the value storage structures. In addition, it reduces the effectiveness of bypass-

† Physical limitations to scaling, while of increasing importance, should not prevent the continuous improvement of CMOS semiconductor technology through at least the end of the decade. Further scaling depends on the development of and transition to non-classical CMOS technologies [45].

ing, which attempts to reduce the impact of register access latency by passing result values directly among execution units. Finally, the emergence of static and dynamic power dissipation as architectural constraints demands efficient implementation of the large amount of value-communication circuitry [16, 65].

The complexity of value communication structures arises from their generality. They are designed to support arbitrary data-dependence relationships among all in-flight instructions. Thus, their complexity depends primarily on details of the *implementation* (e.g., the pipeline width and depth), even as the value communication called for by a given program remains constant. As a result, the overhead of current value communication mechanisms increases as pipelines grow to exploit parallelism.

Moving forward, novel value communication mechanisms must be devised to exploit parallelism more efficiently. Specialized mechanisms, optimized for specific value communication patterns, can fill this role. Rendered independent of the scaling of the pipeline, they will be more resistant to the negative effects of technology trends. A prerequisite for the application of these alternative mechanisms, however, is a way of classifying the communication requirements of each value.

1.2 Degree of Use

The thesis of this work is that a value's *degree of use* provides the most pertinent information regarding that value's communication requirements. Degree of use is simply the number of consumers of a particular register value. It is a dynamic property—that is, successive instantiations of the same instruction in the program may lead to values with different degrees of use. Values that have a high degree of use (i.e., many consumers) must be widely available (e.g., to multiple functional units) at low latencies for long periods of time. In contrast, values that are used but once do not require such a powerful (and expensive) communication mechanism. The best communication mechanism for a value depends on the needs of that value relative to the capabilities of the various available communication mechanisms. The relative proportions of these classes of values will determine the forms of and relative needs for these different mechanisms.

To avoid unwieldy sentences, the degree of use "of an instruction" or "of a register" will be used occasionally throughout this dissertation. Of course, degree of use is a property of a specific

dynamic value. These phrases are shorthand for the degree of use of the value generated by a particular instruction or the degree of use of the value in a particular register, respectively.

1.2.1 An example

Figure 1.1 illustrates some of the interesting properties of degree of use using a short example function that returns the first occupied bucket in a hash table. The C source code example appears on the left and the corresponding Alpha [5] assembly code on the right. A portion of the dynamic dataflow graph corresponding to a particular execution of the function appears at the bottom of the figure.

Alpha assembly code will appear throughout this dissertation, so a brief explanation is provided here. Most instructions with a register destination (e.g., arithmetic and logical operations) name that register on the right. The sole exceptions are load instructions, which name the destination register on the left. All load instruction mnemonics begin with `ld` (the `lda` and `ldah` instructions perform address computations and do not access memory, although their destination registers still appear on the left). Memory addresses for loads and stores appear on the right and consist of a fixed offset to be added to the contents of a base register appearing within parentheses. Control instructions can be identified by the presence of a label, except for the indirect jumps `jmp`, `jsr`, and `ret`, which branch to an address stored in a given register (appearing within parentheses).

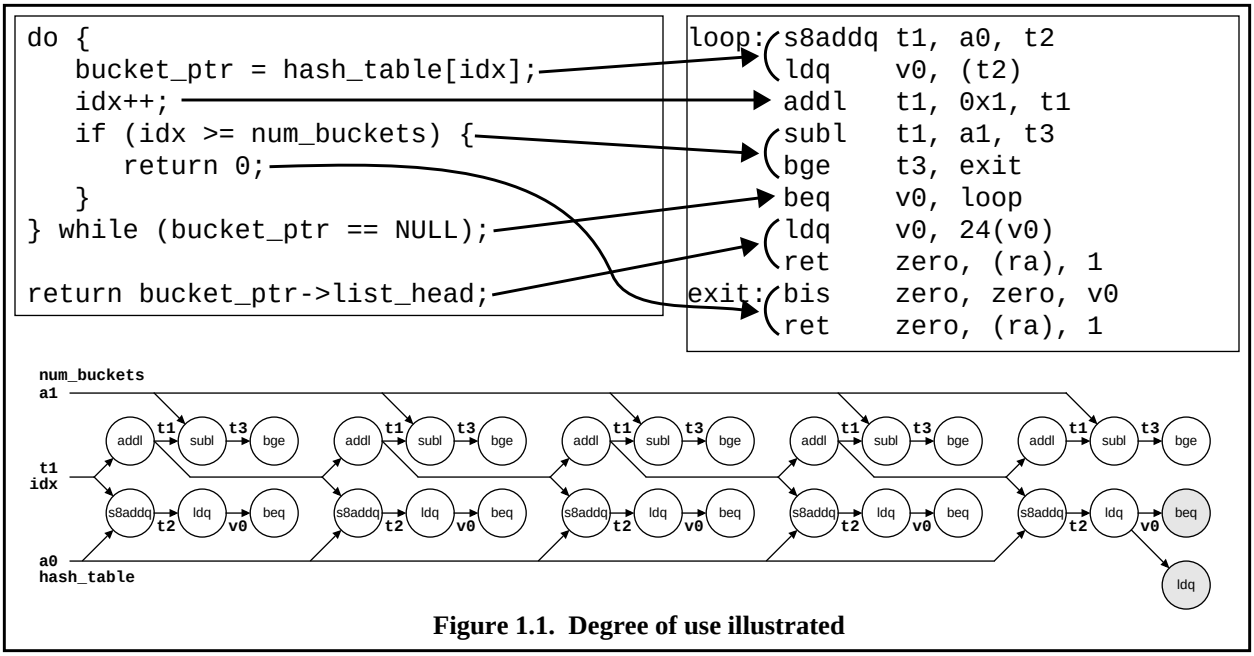


Figure 1.1. Degree of use illustrated

The figure illustrates several interesting aspects of degree of use. First, many values are used only once. Values assigned to `t2` and `t3`, for example, will always exhibit a degree of use of one. Of the 23 values represented in the dataflow graph, 14 of them are used only once. Second, some static instructions always generate values with the same degree of use (e.g., the `s8addq` and `subl` instructions). Instructions whose results are live across one or more conditional branches can have different degrees of use depending on the particular path taken through the program. For example, the value in `t1` has a degree of use of one in the final loop iteration, but a degree of use of three in all prior iterations. Finally, note that it is possible for a value to have a degree of use of zero (i.e., never be used). Had the loop terminated due to the `bge` being taken (rather than the termination of the `while` loop as shown), the two shaded nodes in the dataflow graph would not be executed and the value generated by the `ldq` instruction would not be used.†

1.2.2 Classifying values using degree of use

The example clearly demonstrates how degree of use provides information about the communication of a dynamic value lost as a consequence of the programming model. The level of indirection inherent to register-based architectures obscures this information for the sake of compact instruction encoding. Thus, the ultimate fate of an instruction's result is hidden when that result is generated: it may be read from the destination register once or many million times or any number in between. Without additional information, all value-producing instructions have no choice but to place their results into the specified registers. Also, the microarchitecture must ensure that all registers can support being read any number of times. These consequences are at the root of the inefficiencies in current inter-instruction communication mechanisms.

Degree of use information restores knowledge about the fates of individual dynamic values. Using this knowledge, it is possible to identify values that have different communication needs and handle them in an optimized manner. In fact, degree of use offers the most direct measure of communication, as it quantifies the actual flow of a value to its consumers. Therefore, it is a property solely of the value's dataflow (and, by extension, the value's role in the program). Possible

---

† Because of the focus on register value communication, values are not tracked through memory. Store instructions are terminal: from the perspective of register values, a store represents one use of each of two register values (one containing an address and another the data to be written to memory). Subsequent loads of this stored value would each be considered to create a new value.

alternative measures are influenced by other factors than the communication structure of the program, such as dynamic events. For example, consider classifying a value's communication by its lifetime: values generated by the same instruction and communicated in the same manner may have significantly different lifetimes in consecutive executions due to an unrelated L2 cache miss.

1.2.3 The need for prediction

Using a value's degree of use to optimize its communication requires knowledge of the future. The degree of use of a particular value is calculated by counting its uses; thus, it is not certain until the last use has been observed. Prediction is a well-understood technique for obtaining information that would not normally be available (e.g., the direction of a conditional branch prior to its execution). Based on the predicted information, actions can be taken speculatively, and the prediction verified later. Given high enough prediction accuracy, the effect is the same as having future knowledge.

This work describes *degree of use prediction*, which supplies high-accuracy speculative degree of use information for values early in the processing of their producer instructions. Degree of use prediction is successful because the dataflow patterns in programs are predictable. As demonstrated by the example, many instructions generate values that always have the same number of uses. Determining the degrees of use for values from these instructions is trivial. Even where control flow can affect the degree of use of values from a particular instruction, the predictability of control flow (demonstrated by the success of branch prediction) leads to similar predictability in the degree of use. As with branch prediction, past behavior is a very good predictor of the future.

1.3 Contributions

The initial work on degree of use arose in connection with the Multiscalar project [78]. In this work, Franklin and Sohi presented the distribution of degree of use (coining that term in the process) of dynamic values [32]; they observed the dominance of single-use values and used their observations to propose an alternative value-communication mechanism—namely, a distributed register file. These observations (low average number of uses, many single-use values) have since been used to justify certain design decisions or motivate complete optimizations (e.g., [8, 24, 43,

50]), but none has yet exploited knowledge of number of uses of a particular value because of the unavailability of this information. This dissertation addresses how to obtain that information, its relationship to the role of a value within the program, and how it can be exploited.

The first major contribution of this work is a thorough exploration of the degree of use properties of register values. The observations of Franklin and Sohi are confirmed and expanded upon; the use of a different architecture as well as different benchmarks and compilers broaden the scope of their initial study. Correlations between the role of a value within a program and its degree of use help to explain some of the consistency that is observed with respect to the distribution of different degrees of use. An in-depth characterization of the stability of the degree of use of individual instructions demonstrates the feasibility of history-based dynamic prediction schemes. Finally, the distribution of values with different degrees of use is examined analytically. A previously-proposed analytical model is extended, and the mean degree of use is derived from considerations of the instruction mix.

The demonstration of accurate static and dynamic degree of use prediction is the second contribution of this work. The degree of use of a value is determined by its role in the program; since the role of a given instruction of the program is fixed, the possible degrees of use of values produced by that instruction are pre-determined. A formulation of the degree of use dataflow problem is presented, which, when solved with standard interprocedural dataflow analysis techniques, allows assignment of the set of possible degrees of use to each static instruction of the program. This assignment is the basis for static degree of use prediction. Dynamic degree of use prediction, on the other hand, is based on run-time profiling of degree of use. Here, the novel concept of forward control flow is introduced, which offers short-range path look-ahead as a direct consequence of pipelining. In more advanced dynamic predictors, this information can be used to distinguish among different previously observed degrees of use for the same instruction.

The third significant contribution of this work is *useless instruction elimination* (UIE). This technique avoids the execution of instructions that produce values that will not be used (i.e., degree of use zero values). UIE is representative of the class of optimizations that uses degree of use information to affect the handling of the producer instruction. The performance benefit of UIE is limited by the frequency of occurrence of useless values and the importance of resource contention. Nonetheless, resource utilization is reduced, which can lead to lower power dissipa-

tion. A minor contribution associated with this optimization is an in-depth study of the existence and properties of the useless instructions, which implicates compiler optimizations in increasing the incidence of useless instructions.

Finally, *use-based register caching* applies degree of use information more broadly. The large, slow register file is replaced by the combination of the bypass network and a small, fast register cache. Degree of use information is used to determine the expiration of a value's usefulness by comparing the number of actual uses with the prediction. Only those values that are useful after bypassing are placed in the register cache, enabling its size advantage over the register file. The register file is relegated to the task of recovery, supplying values that were mistakenly dropped from or never placed in the cache.

1.4 Methodology

This section summarizes aspects of the methodology common to the different experiments conducted. The issues presented here are general in nature; details specific to a given experiment are provided in the associated chapter where necessary. Many additional details too cumbersome or arcane to present within a chapter (or here) are described within the appendix.

1.4.1 Benchmarks

The benchmarks used in all experiments are from the SPEC CPU 2000 suite [80]. Depending on the experiment, data is provided for all benchmarks (26 total) or only the integer benchmarks (12 total). Except where noted, the training inputs provided with the benchmark suite were used. Benchmark binaries generated with different compilers and compiler options were used in some experiments. Descriptions of the compilers and the flags used in each configuration may be found in section A.2 of the appendix. If unspecified for a particular experiment, the binaries used were those compiled with the Compaq/Digital C, C++, and Fortran-90 compilers (`cc`, `cxx`, and `f90` under Digital UNIX 4.0, respectively) with the flags that yielded the best performance on an aggressive (8-wide, deeply-pipelined) simulated machine. All binaries were statically-linked.

1.4.2 Simulators

The majority of the results in this dissertation come from execution-driven simulation of user-level code (system calls are executed on the host machine). Two different simulators were used—

a functional simulator and a timing simulator. All characterization data were gathered using functional simulation of benchmarks executing to completion. Performance data (and other associated results) were obtained using a detailed, parameterized microarchitectural timing model. The parameters varied among experiments; important features of the microarchitecture common to all of the experiments are discussed in the next section. The significant slowdown of the timing simulator versus native execution prohibited simulation of the benchmarks to their completion under the timing simulator (a complete timing simulation of `apsi`, for example, runs for more than a month). Instead, the first four billion instructions of each benchmark were simulated.

1.4.3 Model architecture and microarchitecture

The benchmarks are compiled to an Alpha instruction-set architecture (ISA) target [5]. The Alpha ISA is a sequential, register-based, load-store ISA. Thirty-two each integer and floating-point registers are defined; one of each kind always contains zero. Excepting loads and stores, all instructions operate exclusively on register values. Most instructions have one or two inputs, although conditional move instructions are provided, which effectively have three inputs. Instructions have a maximum of one output.

Where timing simulation is required, a pipelined, out-of-order superscalar microarchitecture with MIPS R10K-style register renaming is assumed [89]. Instructions are scheduled dynamically—subject to resource constraints—from an instruction window as soon as their input operands are available. Multi-cycle execution resources are assumed to be fully-pipelined, and the execution of a dependent operation may begin in the cycle immediately following the completion of its parent.

Loads may issue before older store addresses are completely known and assume both a cache hit and no unknown memory dependence to an older store [72]. A cache miss results in the need to re-issue all operations issued after the load through the signalling of the miss. Loads may bypass their data from older executed, unretired stores with the same latency as a cache hit. A conflicting older store executing after a load results in a pipeline squash and refetch of all operations beginning with the load. A load-dependence predictor is used to delay the issuing of loads that have previously caused such squashes [90].

Instruction sequencing uses separate conditional and indirect branch predictors and a return address stack. Execution proceeds down a wrong-path until the mispredicted control instruction is executed; fetch begins along the correct path in the cycle immediately following the execution of a misprediction; more generally, any pipeline squash completes in a single cycle and fetch resumes the following cycle. There is no limit to the number of outstanding, unresolved branches. The performance of most Alpha implementations are sensitive to code layout with respect to branches and their targets [6]. For this reason, the compilers insert many NOPs. These NOPs are eliminated during instruction fetch—they have no effect other than occupying cache space.

The memory hierarchy consists of three caches and a fixed-latency, infinite memory. Separate L1 instruction caches and data caches are backed by a unified L2 cache. The L1 data cache and the L2 cache are writeback caches. The memory system supports multiple outstanding misses at each level; each level also contains an opportunistic stride-based prefetcher. Data and instruction TLBs are perfect (i.e., not modeled).

1.5 Dissertation Outline

The next four chapters of the dissertation each present a primary contribution in the order described in Section 1.3. Chapter 2 presents an in-depth characterization of degree of use properties. Degree of use prediction is described in Chapter 3. Both static and dynamic prediction methods are presented; the formulation of the degree of use dataflow problem appears in connection with the static prediction. Chapter 4 presents useless instruction elimination. The incidence and causes of useless instructions are investigated, followed by a description and experimental evaluation of the UIE technique. Use-based register caching is the topic of Chapter 5. The proposed register cache organization is motivated by considering the bypass network the primary value communication mechanism in lieu of the register file. Managing the register cache contents using degree of use information is demonstrated to be superior to previously proposed techniques. Chapter 6 summarizes the contributions detailed in the prior chapters of the dissertation; it also discusses possible additional applications of degree of use information and issues with such optimizations in general. References and the appendix make up the remainder of the document.

Chapter 2

Characterizing Value Degree of Use

This chapter presents a detailed characterization of inter-instruction communication through registers in terms of degree of use. The data presented here serve three functions: (1) to illuminate the inter-instruction communication patterns that occur in programs, (2) to suggest opportunities for communication optimizations, and (3) to demonstrate the feasibility of degree of use prediction.

First, aggregate degree of use properties are presented. Values generated during a program's execution are classified by their degree of use, and the types of communication that are revealed are discussed. In some cases, the role of a value in the operation of the program can be identified by the type of instruction that generates the value or the architectural register to which the value is assigned. With this additional data, values with specific roles in a program are shown to have degree of use characteristics significantly different than the overall average. Next, the stability of the communication is explored on a per-instruction basis. Stable inter-instruction communication suggests that degree of use prediction, the topic of the next chapter, will be successful. Finally, mathematical models for degree of use characteristics are developed, extending some prior work in this area. The results of this analysis allow for both the calculation of the probability of occurrence of a particular degree of use and the estimation of the mean degree of use from the expected instruction mix.

2.1 Degree of Use Characteristics

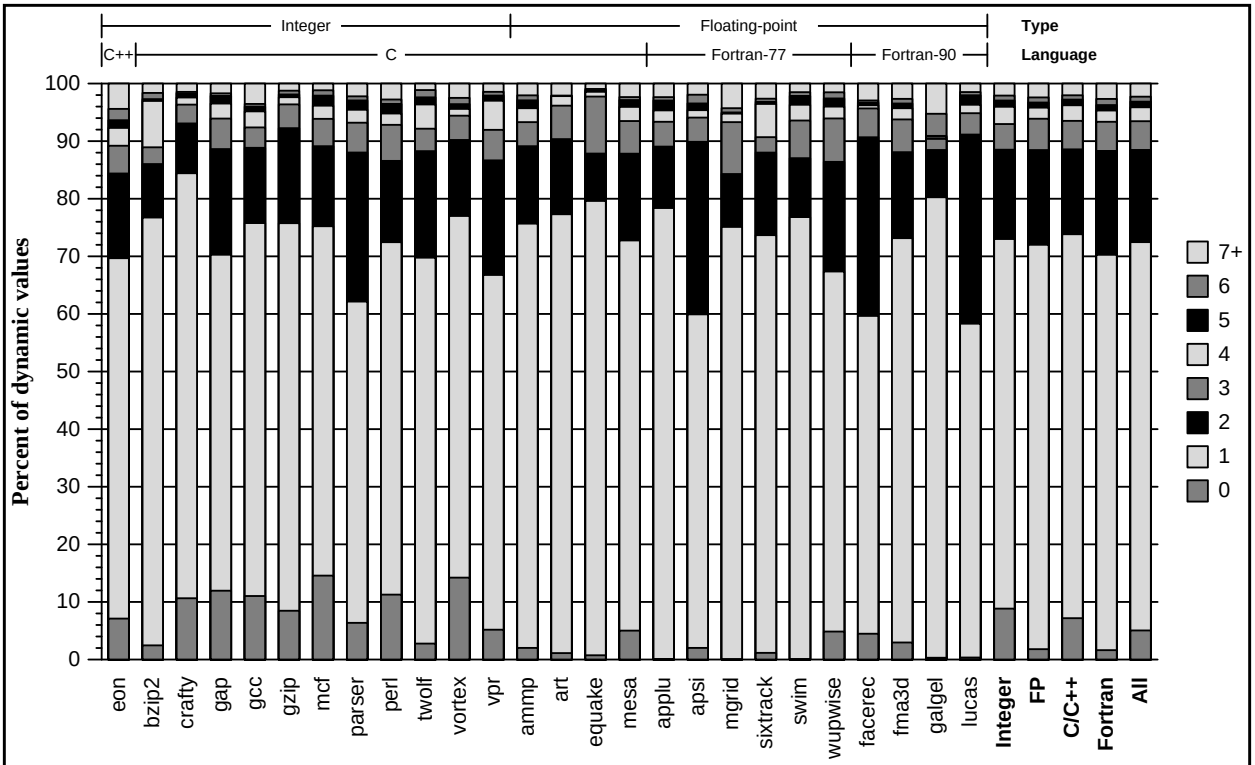
Figure 2.1 shows data on the observed degree of use distribution for each of the benchmarks compiled under two different compiler suites: the vendor suite (offered by the same company responsible for the instruction set architecture) and the third-party suite. Details on the specific compilers in each suite and the compilation methodology can be found in section A.2 of the appendix.

It is readily apparent that most of the communication occurring during program execution is direct communication to one consumer: an average of over 67% of dynamic values have a degree of use of one. The frequency of degree of use two values show the most absolute variation, accounting for anywhere between 4% and 33% of all values. No higher degree of use accounts for more than 11% of the values in any of the benchmarks. 4-5% of the values generated by most of the integer benchmarks are not used at all.

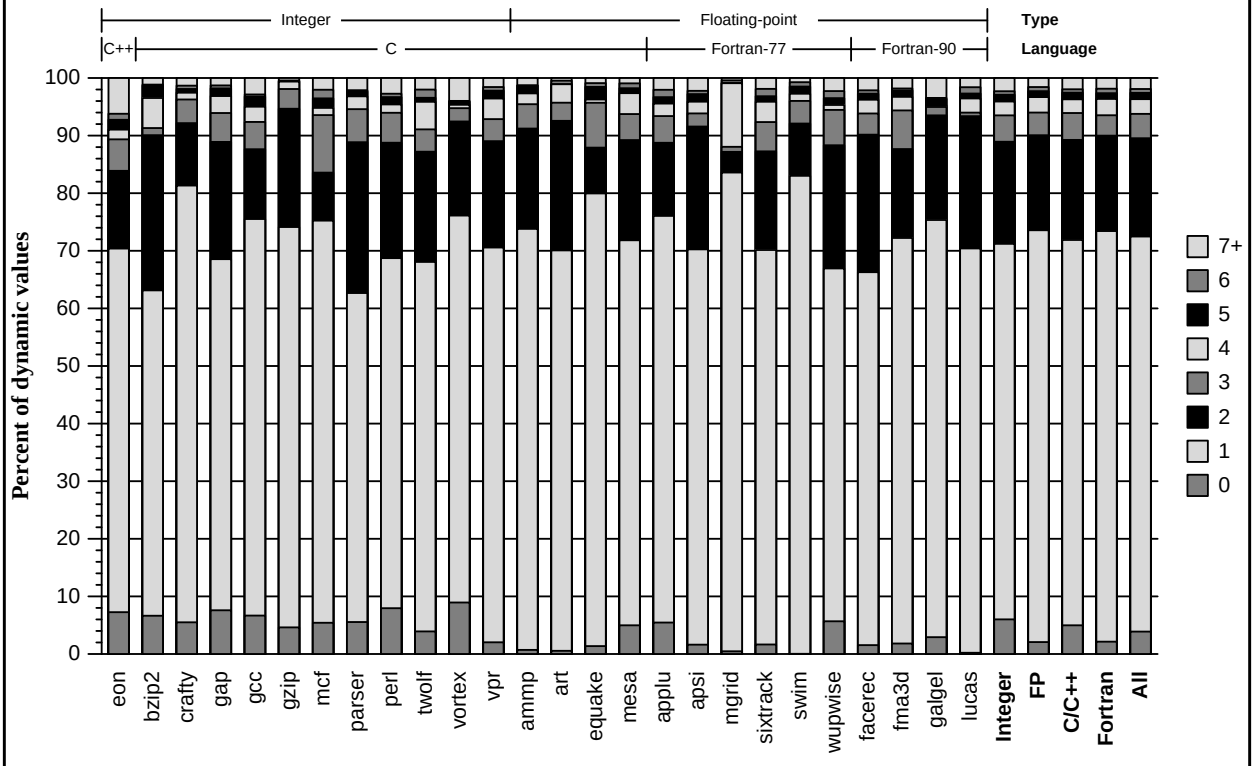
Comparing Figure 2.1(a) and Figure 2.1(b), it is apparent that the degree of use behavior of a given program is quite similar even when different compilers are used, especially for the C and C++ benchmarks. This should not be surprising since the overall value communication structure is a function of the ISA and the program itself. The only immediately obvious difference is a trade-off between one- and two-use values: the vendor compilers tend to generate more of the former and less of the latter while the third-party compilers do the opposite. The third-party compilers also generate fewer degree of use zero instructions. These minute differences arise from how each compiler performs register allocation, code scheduling, and other optimizations that affect how registers are mapped onto the inherent dataflow specified by the program.

2.1.1 Mean, median, mode, and maximum

The median and mode degrees of use are easily observed on each distribution of Figure 2.1 (the bin crossed by the 50% level and the largest bin, respectively). Without exception, the median and mode degree of use are one. The mean degree of use for each benchmark appears in Table 2.1 along with the maximum degree of use and the percentage of non-nop instructions that produce a register result. An average of about 76% of dynamic instructions produce a value; the remainder are almost entirely stores and branches, although a few rare instructions (e.g., certain system



(a) Vendor Compiler Suite



(b) Third-party Compiler Suite

Figure 2.1. Observed degree of use

Table 2.1: Degree of Use Characteristics

	Vendor Compiler Suite			Third-party Compiler Suite		
Benchmark	Mean	Maximum	% Dynamic	Mean	Maximum	% Dynamic
`eon`	1.94	8.46 K	63.7%	2.06	8.49 K	62.4%
`bzip2`	1.76	336 M	76.6%	1.74	67.5 M	81.1%
`crafty`	1.53	5.40 M	82.8%	1.54	2.34 M	76.5%
`gap`	1.53	3.21 M	76.0%	1.59	11.6 M	72.7%
`gcc`	1.70	2.68 M	69.7%	1.73	421 K	62.5%
`gzip`	1.58	828 K	76.2%	1.52	4.19 M	78.0%
`mcf`	1.51	34.0 M	67.5%	1.71	33.8 M	70.4%
`parser`	1.76	13.8 M	74.2%	1.73	3.26 M	71.4%
`perl`	1.68	579 K	71.0%	1.67	122 K	67.4%
`twolf`	1.68	69.3 K	70.8%	1.78	44.1 K	72.9%
`vortex`	1.54	6.98 M	71.4%	1.61	86.7 K	66.1%
`vpr`	1.72	28.5 M	71.7%	1.63	864 K	77.9%
`ammp`	1.88	205 M	81.8%	1.78	86.6 M	82.9%
`art`	1.96	250 K	67.3%	1.69	57.1 M	81.5%
`equake`	1.56	7.66 M	92.1%	1.58	3.57 M	86.6%
`mesa`	1.72	1.31 M	74.9%	1.66	1.31 M	74.2%
`applu`	1.91	1.67 M	86.1%	1.78	2.08 M	65.7%
`apsi`	1.92	6.25 M	80.9%	1.85	2.50 M	78.2%
`mgrid`	1.74	344 K	91.9%	1.59	3.44 M	80.8%
`sixtrack`	1.91	16.3 M	88.3%	1.99	5.64 M	73.9%
`swim`	1.85	863 K	82.8%	1.63	967 K	74.2%
`wupwise`	1.73	2.89 M	79.1%	1.78	9.22 M	67.3%
`facerec`	1.85	11.3 M	82.2%	1.79	2.96 M	78.4%
`fma3d`	1.90	80.3 K	77.9%	1.80	41.8 K	79.9%
`galgel`	1.85	2.99 M	75.9%	1.75	6.73 M	78.8%
`lucas`	1.83	51.5 K	90.4%	1.80	24.9 M	75.9%
Integer	1.66	336 M	72.6%	1.69	67.5 M	71.6%
Floating Pt.	1.83	205 M	82.3%	1.75	86.6 M	77.0%
C/C++	1.69	336 M	74.2%	1.69	86.6 M	74.0%
Fortran	1.85	16.3 M	83.6%	1.78	24.9 M	75.3%
All	1.75	336 M	77.8%	1.72	67.5 M	74.5%

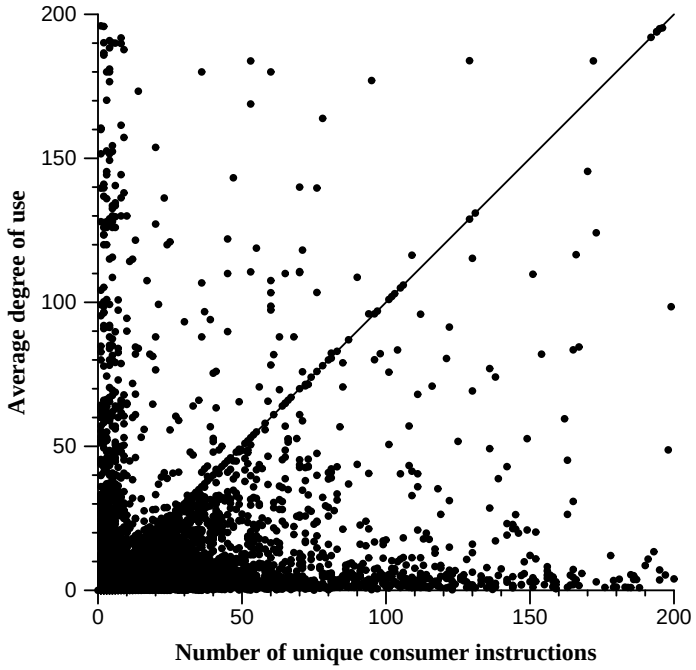


Figure 2.2. Correlation between number of potential static consumers and degree of use

calls) do not produce results either. Note that the average degree of use differs very little between the different compiler suites.

While the relative fraction of values with a high degree of use is very small, the data in Table 2.1 show that degrees of use themselves can be large. Maximum degrees of use range over five orders of magnitude from a few thousand to over three hundred million (a global pointer value in `bzip2`) across the different benchmarks. Instructions exhibiting the largest degrees of use fall into two overlapping categories: (1) address-generating instructions (often stack and global pointer updates), and (2) instructions generating loop-invariants. In the first case, the number of unique static consumer instructions tends to be high, while in the second case the high degree of use frequently results from repeated communication to a set of static consumers.

This relationship between the number of unique consumer instructions and the average degree of use appears in Figure 2.2 for static instructions generating a significant number of values.† The diagonal line corresponds to an average degree of use equal to the number of static consumers. Points above this line represent instructions that generate values used within loops: the average

† Only static instructions generating more than 1000 values are represented. The data in the figure corresponds to benchmarks compiled with the vendor compilers; data from the benchmarks compiled with the other compilers appears nearly identical.

number of uses of such a value exceeds the number of unique consumer instructions. A typical example of such an instruction is one that generates the base address of an array accessed in the loop body. Conversely, points below the line represent instructions generating values with a smaller average degree of use than the number of potential consumer instructions. For these instructions, variation in the subsequent control flow results in different consumers receiving the result of the instructions on different executions. Note that some of these consumers may still receive one of these values multiple times due to looping. A representative example of an instruction in this region is an indirect subroutine call, which has as its result the return address: many return instructions in different subroutines will use the result of the call, but the average degree of use will be only one.

2.1.2 Degree of use of input values

The average degree of use of instruction inputs is higher than the average degree of use of instruction results. To understand this phenomenon, consider a single value used ten times. The degree of use of the value is ten and that value would be counted once in the distributions of Figure 2.1. However, that value accounts for *ten* instruction inputs: while only one instruction generates a result with a degree of use of ten, ten instructions *use* a degree-of-use-ten value as an input. In other words, the frequency of occurrence of a particular degree of use as an input is its frequency as a result weighted by the degree of use itself.

Therefore, the distribution of the degree of use of instruction inputs may be obtained from the distribution in Figure 2.1(a) by doubling the height of the degree of use two bar, tripling the height of the degree of use three bar, and so on, and then renormalizing. The resulting distribution is portrayed in Figure 2.3 (for benchmarks compiled with the vendor compilers). Note that while an instruction is most likely to *generate* a value that is used once, an instruction will most likely *use* a value that is used more than once. Values with seven or more uses account for nearly a quarter of all values used even though they comprise less than 3% of all values (see Figure 2.1(a)). As would be expected, degree of use zero values account for no inputs.

Another way to think about distributions of Figure 2.3 is as the relative contribution that values with a particular degree of use make to the average degree of use. Consider how the mean degree of use is calculated: each degree of use is multiplied by its frequency of occurrence to

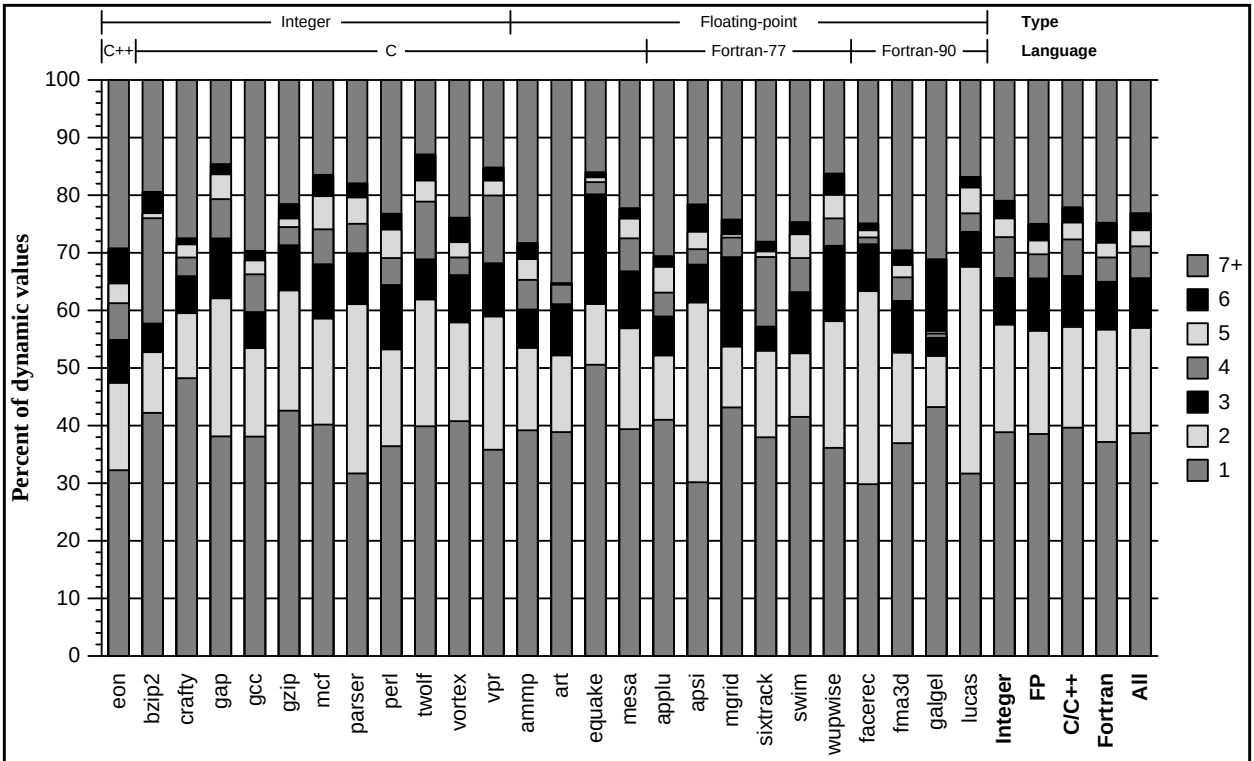


Figure 2.3. Degree of use of instruction input values

obtain its contribution to the mean (an exercise that will be demonstrated more explicitly in Section 2.5). This procedure is exactly that used to generate the distribution of input degrees of use (except for the final renormalization). Figure 2.3 shows that while single-use values comprise the majority (just over two-thirds) of all values generated, they contribute less than 40% to the overall mean degree of use.

2.2 Classifying Values

The degree of use of a value is intimately tied to the role of that value in the execution of a program. Therefore, one expects to see different degree of use properties for different classes of values. In this section, two proxies for the role of a value in a program are considered: the type of instruction that generates the value and the architectural register to which the value is assigned.

2.2.1 Classification by instruction type

Table 2.2 shows degree of use data for six classes of instructions. In addition to the average degree of use, the table shows the average number of unique static consumers for that class of

instruction. The categories in the table represent about 80% of dynamic instructions, including the largest five (control instructions were among the smallest classes of value-generating instructions). The categories listed were chosen based on whether the degree of use data were noteworthy or could be explained by considering the role of those types of instructions within a program.

Load instructions exhibit a larger than average number of static consumers although their degree of use is about equal to the overall average. Further partitioning the loads by the size of the memory access reveals that 64-bit integer loads average over six distinct consumers, while the remaining loads average less than two. In spite of this distinction, the average degree of use of 64-bit loads is only 13% higher than that of other types. Due to the variety of roles for loads in a program, it is difficult to draw any conclusions from these characteristics, although one might speculate that the behavior of the 64-bit loads is correlated with the fact that addresses are 64 bits.

The data in the table indicate a significant difference in the behavior of the results of integer and floating-point math operations. These categories include add, subtract, multiply, divide, and square root instructions of all data sizes but exclude scaled adds, shifts, conversions, and other logical operations, which showed somewhat different behavior. The data show that integer values are used more often than floating-point values; besides their role in computation, integer values perform other functions that are likely to result in many uses of the same value. For example, some integer variables are loop induction variables or containers of multiple smaller-sized data items (e.g., packed bytes or bitmasks). Variation in control flow causes the number of potential consumers to exceed the average number of uses for integer results; floating-point operations typically communicate to a fixed set of consumers.

The `lda` or load address instruction exhibited the highest average degree of use of any common instruction. `lda` places the result of an effective address calculation into a register where it often is used as a base address for many loads or stores, accounting for both the high average degree of use and number of unique consumer instructions. Those `lda` instructions that did not take a register input were classified instead as load immediates, which exhibit very different behavior.

The results of compare instructions have a very well-defined role—determining the direction of a subsequent conditional branch. These values are almost always used just once by a subsequent branch. Occasionally, a branch condition will be computed by a compare instruction prior

Table 2.2: Degree of Use Properties of Instruction Groups

	Vendor Compiler Suite			Third-party Compiler Suite		
Group	% dynamic instructions	Average consumers	Average DOU	% dynamic instructions	Average consumers	Average DOU
Load	30.3%	3.73	1.71	31.0%	3.71	1.67
FP math	22.1%	1.46	1.34	18.3%	2.00	1.70
Integer math	11.0%	2.61	1.74	13.2%	2.84	1.77
Load address	9.6%	11.42	4.90	8.6%	16.47	6.00
Compare	5.0%	1.14	1.04	6.2%	1.15	1.01
Control	1.1%	3.12	1.18	1.0%	2.93	1.08

to one or more different branches, resulting in more than one possible consumer for the computed condition. This phenomenon accounts for the number of potential consumers exceeding one.

Finally, control instructions (those that generate values) supply many different consumers, but their average degree of use is much lower than that of the load address instructions. This category is almost entirely composed of subroutine call instructions, each of which generates a return address used by one of many potential consumers. Indirect calls (`jsr`), comprising about 10% of this category, have an average of 75 consumers, while the more common direct calls (`bsr`) supply an average of only 1.3 consumers.

2.2.2 Classification by register type

Guessing the role of a value using only the nature of its source instruction is difficult. Loads, for example, perform too many different functions in a program to hint at the purpose of the loaded value. A better clue to the role of a value is the identity of the architectural register containing that value. Many registers or groups of registers have functions designated by convention to allow different software components (e.g., compilers, assemblers, and libraries) to cooperate. The functions of these registers should correlate with the degree of use properties of the values assigned to them. The stack pointer is perhaps the most obvious example: within a function body, accesses to local variables stored in the stack frame use the stack pointer as a base register. Thus, one expects values in this register to exhibit a high average degree of use, which is the case.

Table 2.3 shows the average degree of use for several classes of registers (as defined by the Alpha Assembly Language Programmers Guide [6]). Data are shown for benchmarks compiled with the vendor compilers only; the other compiler suite exhibits comparable behavior.

Of all registers or groups of registers, the stack pointer has the highest average degree of use. Within Fortran programs, which frequently pass large arrays on the stack, stack pointer values are used almost three times as often as in the C or C++ programs. The global pointer, used as a base address for the access of global data objects, also shows a significantly higher average degree of use than other register categories, although the behavior is much more consistent across the different benchmark groups.

Callee-saved integer and floating-point registers are those integer registers defined by the assembly language programming conventions to be preserved across subroutine calls. If a subroutine uses one of these registers, it must save the register first and restore it before returning. The higher degree of use of these registers may be ascribed to two factors. First, a smart compiler will preferentially assign values with long live ranges to these registers to avoid having to spill them before subroutine calls. These long-lived values are more likely to have a high degree of use (see Section 2.3). Second, the save operation (a store into the stack frame) results in a use of the value not demanded by the underlying communication structure of the program. Thus, a save results in an additional use being credited to the a callee-saved register value. If the value happens to be dead at the time of a call, the restore at the end of the subroutine will create a value that will

Table 2.3: Average Degree of Use by Register Class

Benchmarks	Stack pointer	Global pointer	Callee-saved[a]	Temporaries[b]	Special integer[c]	FP callee-saved[d]	All other FP
C/C++	7.65	5.09	1.77	1.44	1.15	2.00	1.24
Fortran	30.11	14.09	3.62	2.13	1.69	1.63	1.47
Integer	8.13	5.05	1.94	1.48	1.18	1.45	1.39
FP	18.48	9.59	3.32	1.99	1.55	1.61	1.46
All	11.63	6.05	2.53	1.71	1.38	1.61	1.46

a. `s0-5`
b. `v0`, `a0-5`, `t0-11`
c. `t12`, `at`, `ra`
d. `f2-9`

not be used. In Chapter 4, such restore instructions are shown to account for a significant fraction of the degree of use zero values observed.

Values in integer registers have a higher average degree of use than those in the floating-point registers, matching the behavior observed in the results of integer and floating-point math operations (see Table 2.2). Interestingly, this is true for both integer and floating-point programs. In fact, the higher average degree of use of floating-point programs (see Table 2.1) is due to a greater average degree of use among the *integer* register values in those programs. This effect can be attributed to the existence of many high-use loop induction and array base address variables in the numerical codes.

Temporaries are those registers frequently used by the compiler during general register allocation and expression evaluation (i.e., as intermediates in the evaluation of a complicated source expression). In the integer programs, the average degree of use of values in these registers is under the overall average (from Table 2.1) because many of them are involved in the aforementioned expression evaluation, which tends to result in single-use values. Floating-point benchmarks use many integer temporaries for addresses, which inflates their average degree of use.

2.3 Temporal Characteristics

While degree of use of a value is not a temporal property (i.e., it is not specifically a timing related parameter), it does have a temporal aspect. In particular, it is interesting to correlate a value's degree of use with the intervals between the value's generation and its first use, last use, and overwrite. Each of these intervals has a particular importance. The first use of a value begins the communication process; knowing the distance between the value's creation and its first use defines a window in which communication can be detected. Similarly, the distance to the last use defines a window that contains the entire communication of a value. It also indicates the lifetime of the value from the perspective of the program. Finally, the distance between consecutive definitions of the same register indicates the lifetime of the value from the perspective of the hardware. Absent some additional information, observing the definition of a register is the only signal that there will be no more uses of the value previously occupying that register.

Figure 2.4 illustrates how these intervals are correlated with a value's degree of use. To avoid making these measurements dependent on the configuration of a particular machine, the results

are expressed in terms of the number of instructions rather than a number of cycles. The very long-tailed nature of the distance distributions renders the mean distance an inappropriate measure [28]. Therefore, the data plotted in Figure 2.4 are the median distances with all benchmarks in each group weighted equally.

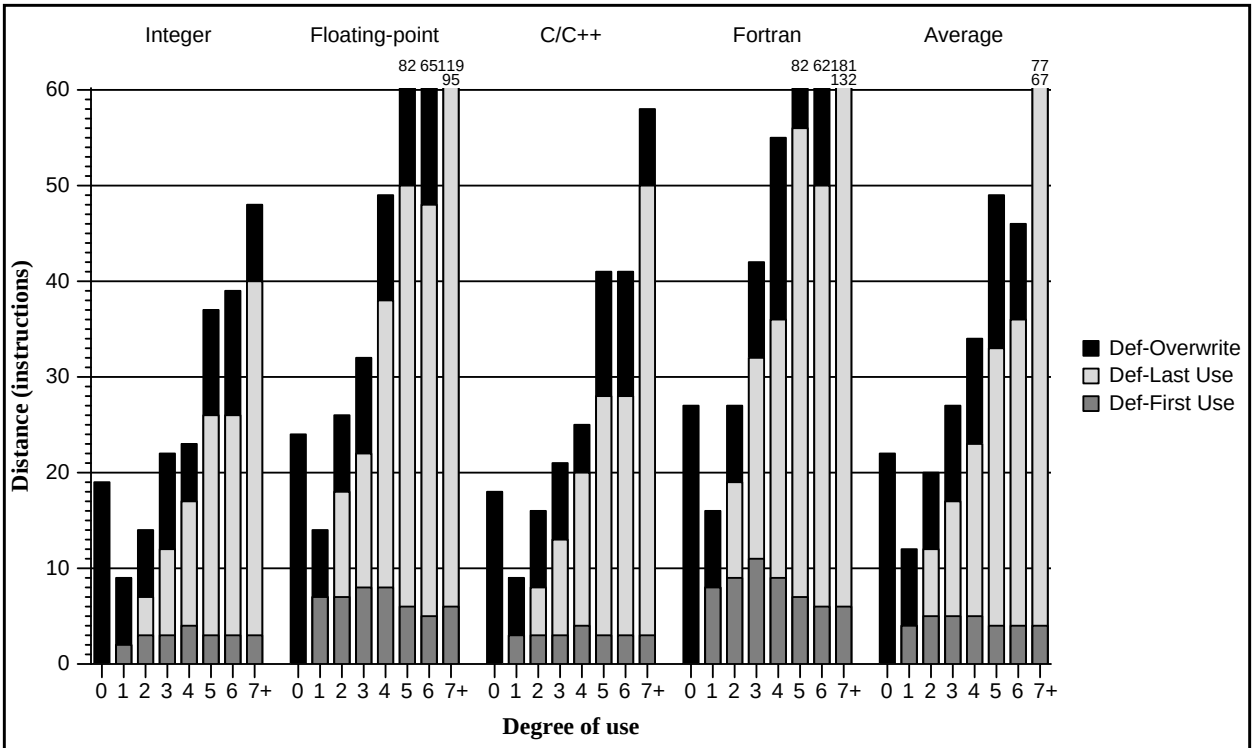
The first use of a value occurs soon after the value's definition, independent of the degree of use. This behavior is expected since one goal of the compiler is to minimize values' live ranges. Programs compiled by the vendor Fortran compiler tend to have more instructions between the generation of a value and its first use than the C/C++ codes. This difference may be attributed to instruction scheduling of multi-cycle latency floating-point operations by this particular compiler, especially within unrolled loops; the scheduler attempts to insert independent instructions between the generation of a value by a long-latency operation and its subsequent use.

The interval between the generation and its final use is correlated with the degree of use of a value—higher degrees of use require more instructions to reach the final use. The correlation breaks down in some groups of benchmarks for degrees of use greater than four. Further investigation indicates that this is not systematic; rather, the number of static instructions generating values with high numbers of uses is relatively small and is therefore more subject to influence by a small set of frequently-executed instructions with unrepresentative behavior.

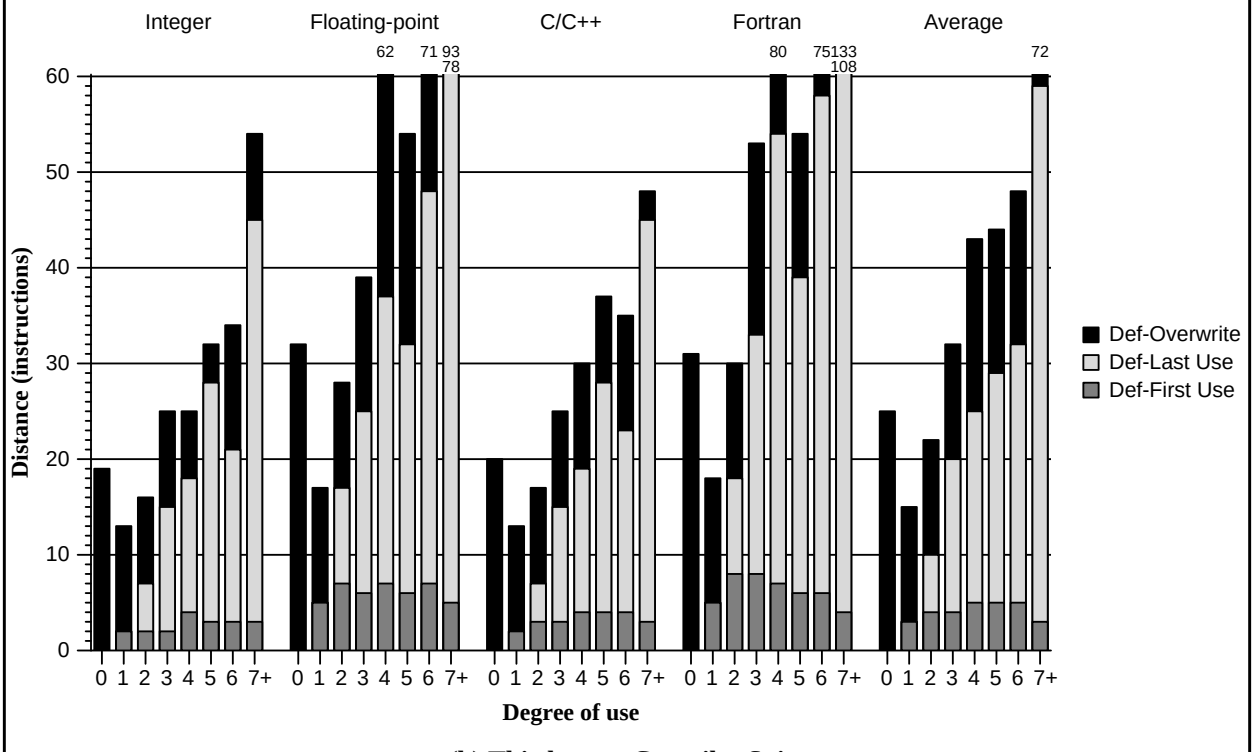
The distance between consecutive definitions of a register tracks degree of use in a manner similar to the distance between the definition and the final use. Note that degree of use zero values exhibit a relatively long interval between the definition of the value and its overwrite, especially for codes compiled with the vendor compilers. It is to be expected that for such values the defining and overwriting instructions are in different basic blocks; otherwise, the definitions could have been trivially optimized away. Values with uses, especially those with just one use, may be overwritten within a single basic block.

2.4 Working Set Behavior

Of particular interest in characterizing inter-instruction communication is the stability of that communication during the execution of a program. Stability of communication is tied to the number of possible degrees of use for values generated by a given static instruction, the relative fre-



(a) Vendor Compiler Suite



(b) Third-party Compiler Suite

Figure 2.4. Distance between a value's generation and its first use, last use, and overwrite

quencies with which those different degrees of use occur, and the temporal locality in the degrees of use of consecutive values from that instruction.

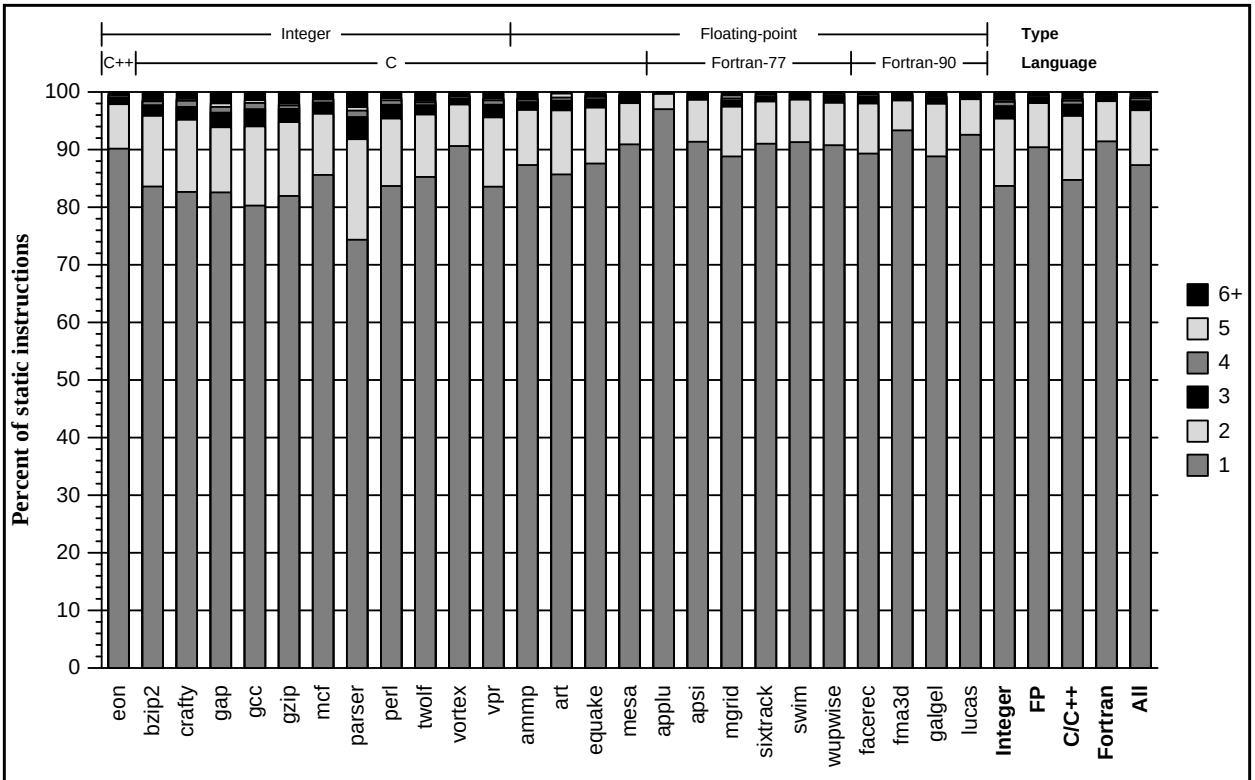
2.4.1 Number of possible degrees of use

Figure 2.5 classifies static instructions based on the number of different degrees of use of values generated by the instructions of the class. The vast majority of static instructions (87% on average) generate the same degree of use every execution. Due to their richer control-flow, integer programs have a slightly larger fraction of instructions that generate multiple degrees of use. Static instructions having more than two unique degrees of use comprise less than 3.5%, and in no case more than 8.5%, of all static instructions.

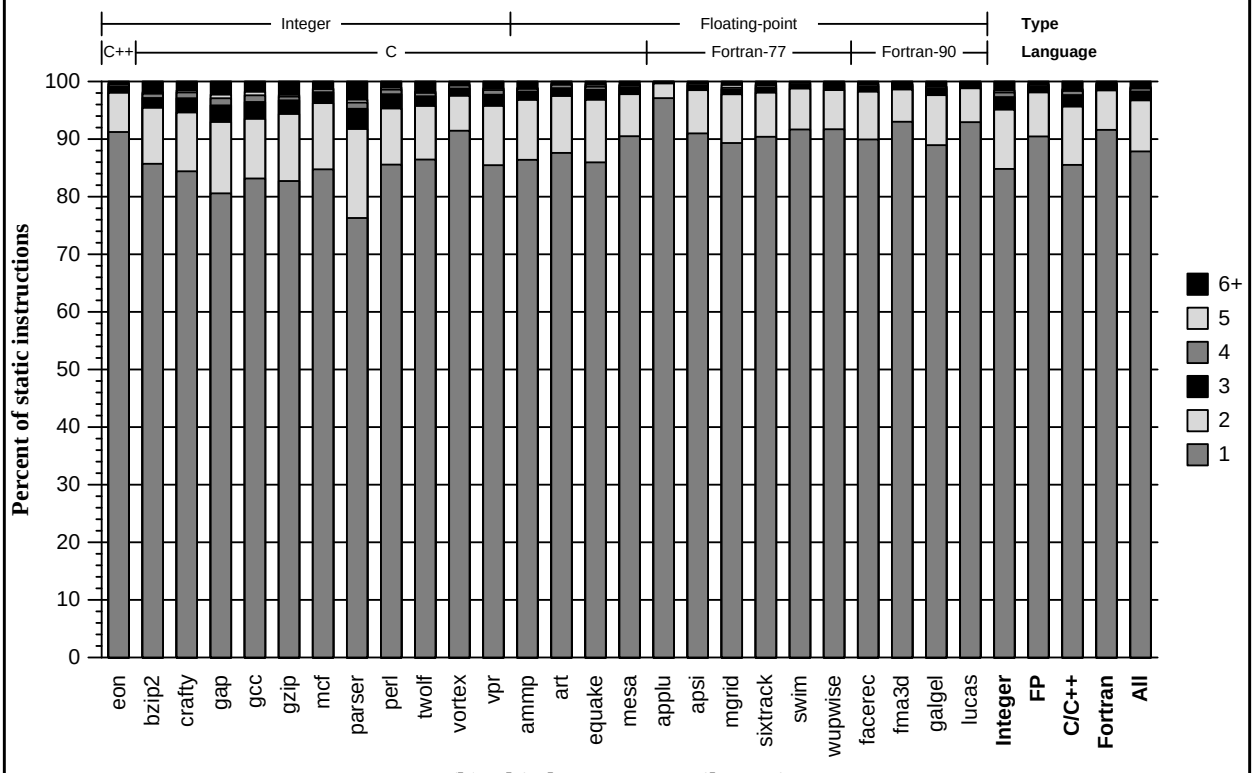
The data of Figure 2.5 pertain to a particular execution of each benchmark. Thus, while a significant number of instructions only generate values with a particular degree of use, it does not mean that these instructions cannot possibly generate values with other degrees of use in other runs of the same benchmark (e.g., given other inputs). The determination of the possible degrees of use for an instruction may be formulated as a dataflow problem. This dataflow problem was solved for these benchmarks to estimate the impact of variations in control flow on the stability of inter-instruction communication. For details on the dataflow formulation of the degree of use problem, see Section 3.3.1.

The results of the analysis appear in Figure 2.6. Static instructions not represented in the data of Figure 2.5 were filtered to allow a more direct comparison between the two figures.† The static analysis data indicate that much more variability in the degree of use is possible than is observed. However, an indeterminate amount of this increased variability results from limitations of the analysis itself. The dataflow analysis is formulated in such a way that it is guaranteed to be safe—no degree of use can occur that is not identified by the analysis. However, it may assign degrees of use to certain instructions that only occur along infeasible paths through the program (e.g., an impossible path through two branches that are correlated). Therefore, the data in Figure 2.6 must be considered to be an upper bound on the variability of the degree of use; the real results will be

† One small difference between the two figures is that the static analysis cannot distinguish multiple degrees of use above six uses. Regardless of how many unique degrees of use greater than six can occur for a particular static instruction, they count for only one unique degree in Figure 2.6. Figure 2.5 does count each degree of use separately, but, due to the multiplicative effects of few high-use values and few many-degree instructions, the difference between the two methods is negligible.

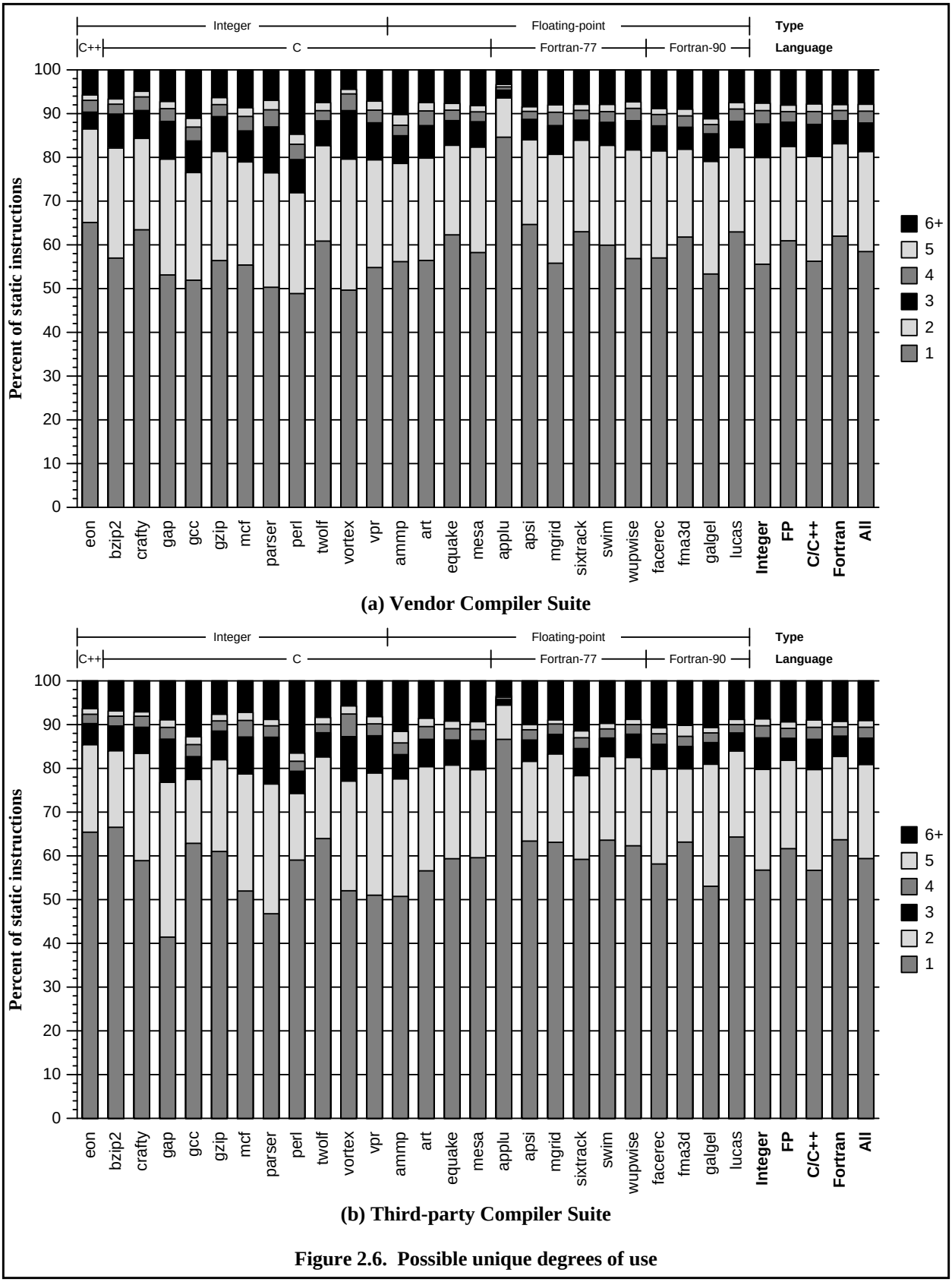


(a) Vendor Compiler Suite



(b) Third-party Compiler Suite

Figure 2.5. Unique degrees of use



(a) Vendor Compiler Suite

(b) Third-party Compiler Suite

Figure 2.6. Possible unique degrees of use

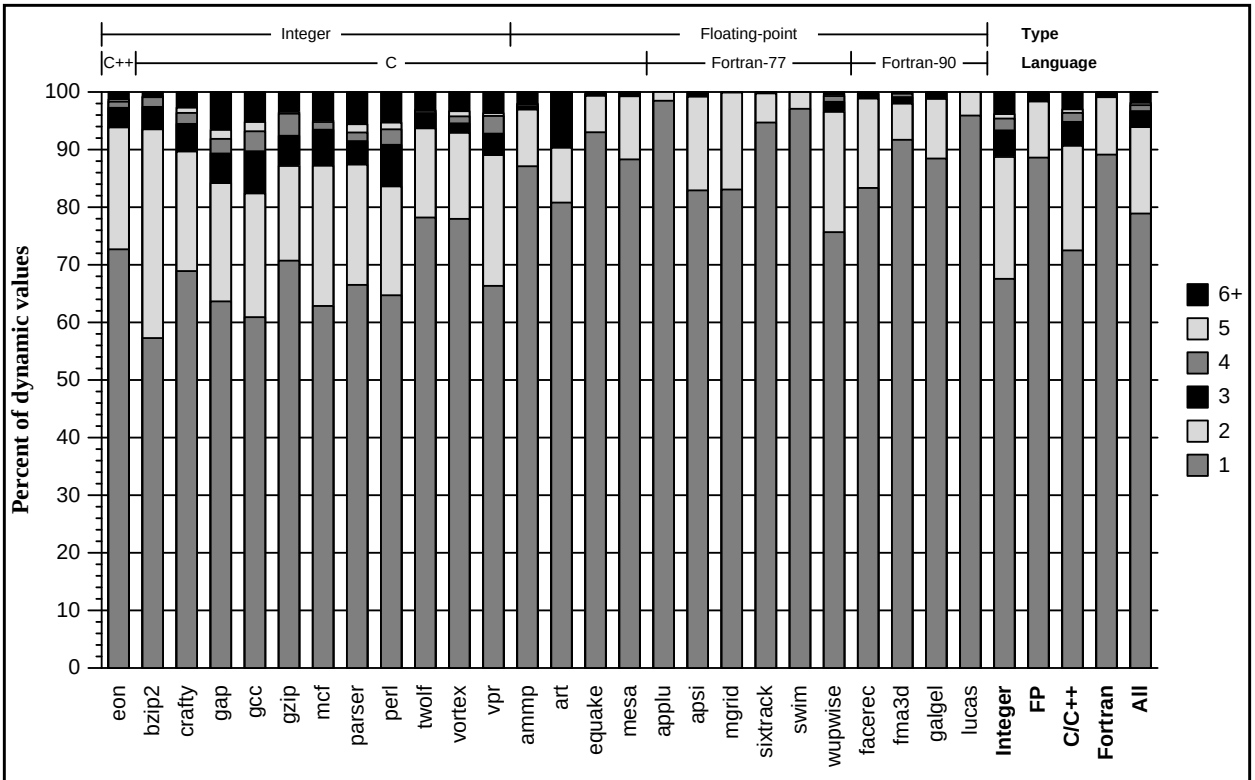
in between those in Figure 2.5 and Figure 2.6. Even this conservative analysis indicates that some 58% of static instructions are constrained to generate only a single degree of use; less than 10% of them generate more than four.

Figure 2.7 presents the distributions of Figure 2.5 weighted by the execution counts of the static instructions. In other words, these distributions show the probability that a random dynamic value originated from a static instruction capable of generating the indicated number of different degrees of use. On average, 14% of the value-producing static instructions in these benchmarks are executed only once, resulting in an underestimation of the importance of instructions with more variable behavior in Figure 2.6. The data in Figure 2.7 indicate that while instructions with multiple possible degrees of use can generate as many as 43% of all values generated by a program, they account for only 21% on average. Less than 4% of values are produced by instructions with more than three different degrees of use.

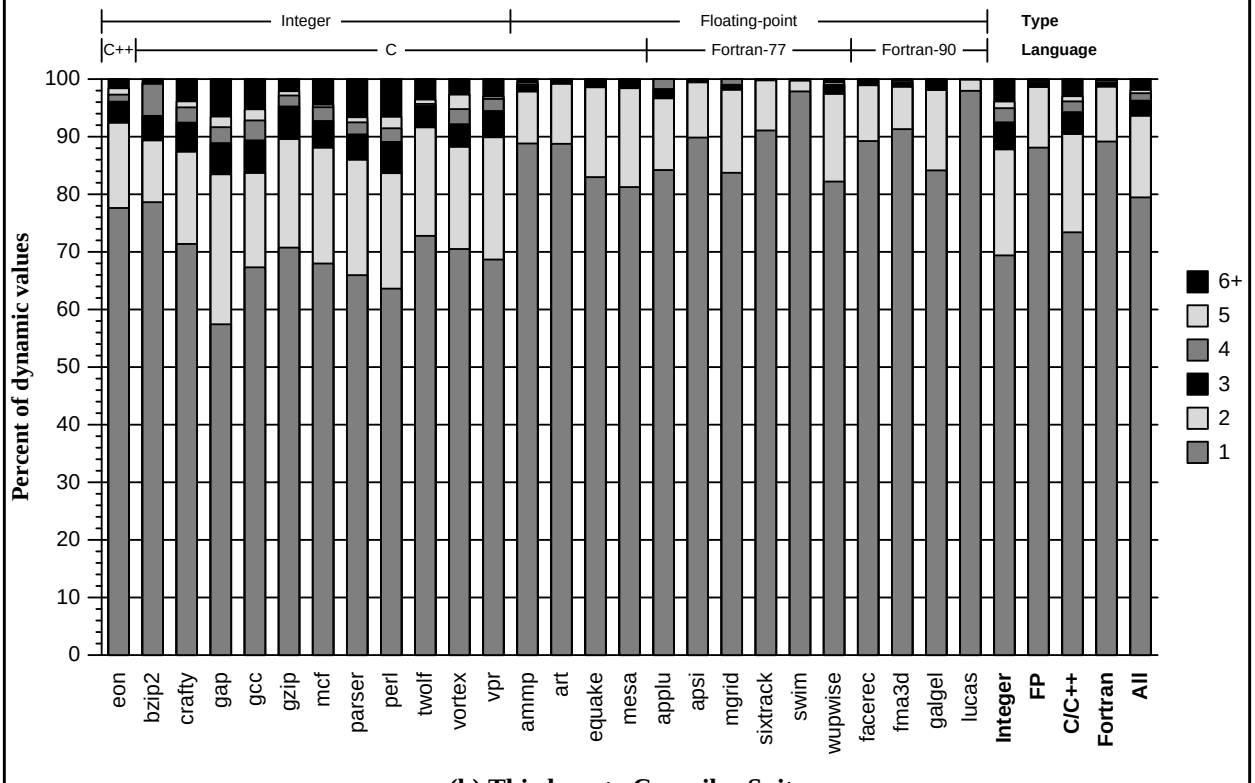
2.4.2 Relative likelihood of possible degrees of use

Just because an instruction generates values with many different degrees of use does not imply that each of the possible degrees of use is equally likely. A static instruction executed one million times might generate values with the same degree of use 99% of those times or it might generate values alternating between two different degrees of use. These behaviors are obviously very different, but that difference is not captured in the data of Figure 2.5 or Figure 2.7. Figure 2.8 presents a distribution of dynamic values based on the frequency of occurrence of values with the same degree of use from the same static instruction. In other words, the distribution bin *N* represents the frequency that a given value has the *N*th most common degree of use of its originating instruction. Over 96% of values exhibit whatever degree of use is most common for their producer instructions. The two most common degrees of use possible from each static instruction account for more than 98% of values for even the worst-case benchmark and for 99.5% of values overall.

Figure 2.9 presents another slice of the same data depicted in Figure 2.8. In Figure 2.9, the data from groups of benchmarks are combined and then broken down by the number of unique degrees of use possible. Obviously, all values from instructions generating only one unique degree of use have the most common degree of use possible from that instruction. About 90% of

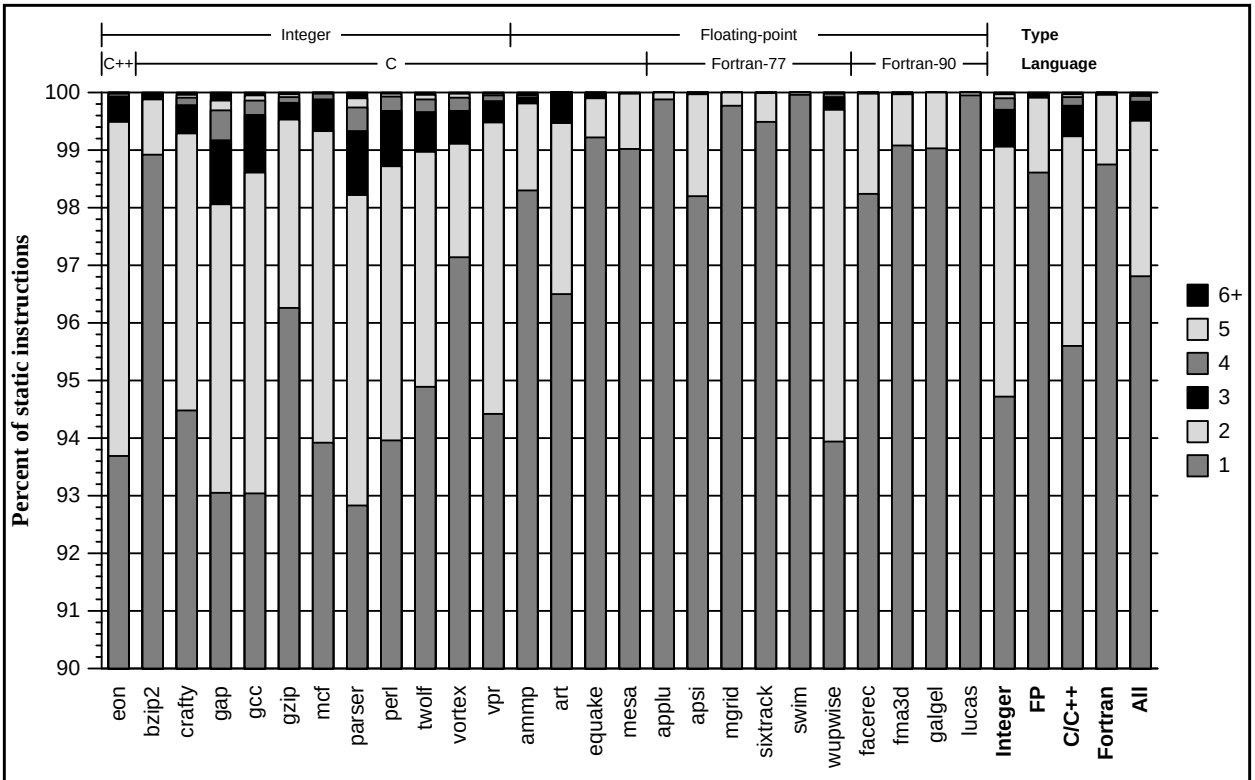


(a) Vendor Compiler Suite

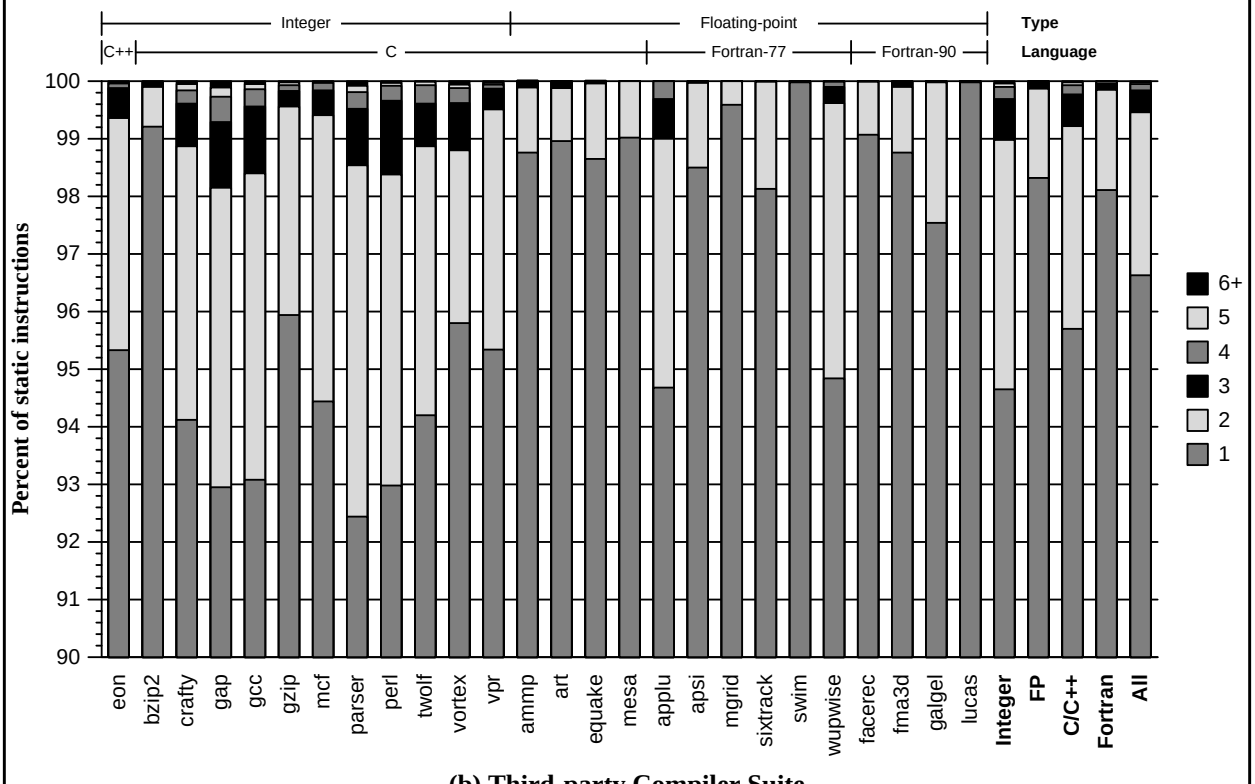


(b) Third-party Compiler Suite

Figure 2.7. Unique degrees of use weighted by execution count

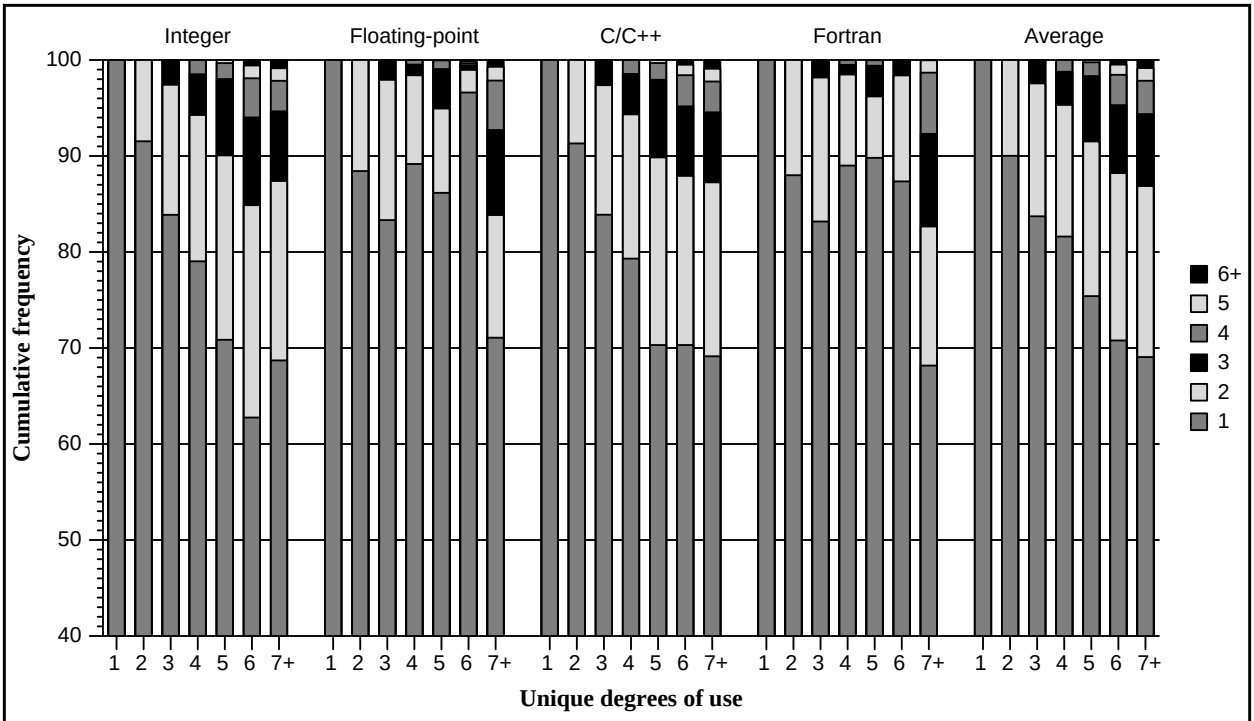


(a) Vendor Compiler Suite

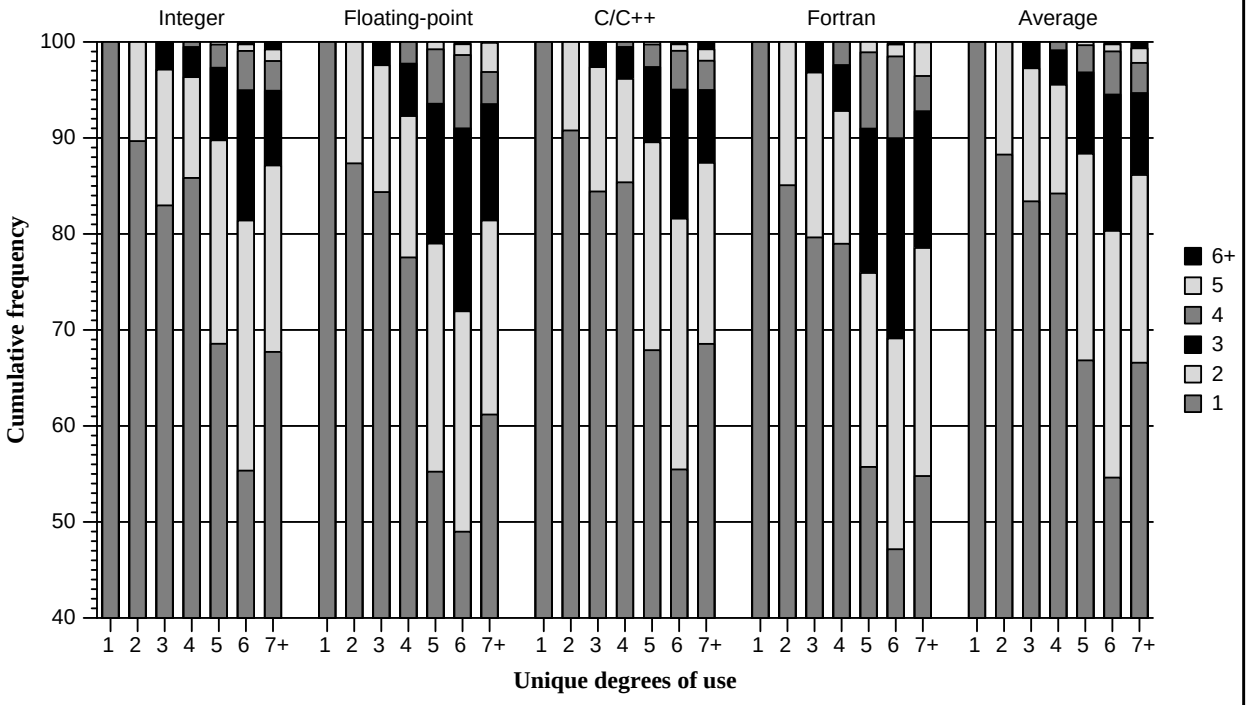


(b) Third-party Compiler Suite

Figure 2.8. Likelihood of possible degrees of use from static instructions



(a) Vendor Compiler Suite



(b) Third-party Compiler Suite

Figure 2.9. Likelihood of possible degrees of use from static instructions

values from instructions with two potential degrees of use end up with the most common of the two. The figure illustrates that even among those instructions that can generate values with many different degrees of use, one particular degree of use is dominant.

2.4.3 Temporal locality in per-instruction degrees of use

Finally, it is important to ascertain whether there is any temporal locality to the degrees of use generated by a particular static instruction. Consider an instruction that generates two different degrees of use with equal frequency over the execution of a program. If these different degrees of use are distributed randomly, this makes prediction more difficult than if all instances with one degree of use occur sequentially before those generating the other possible degree of use.

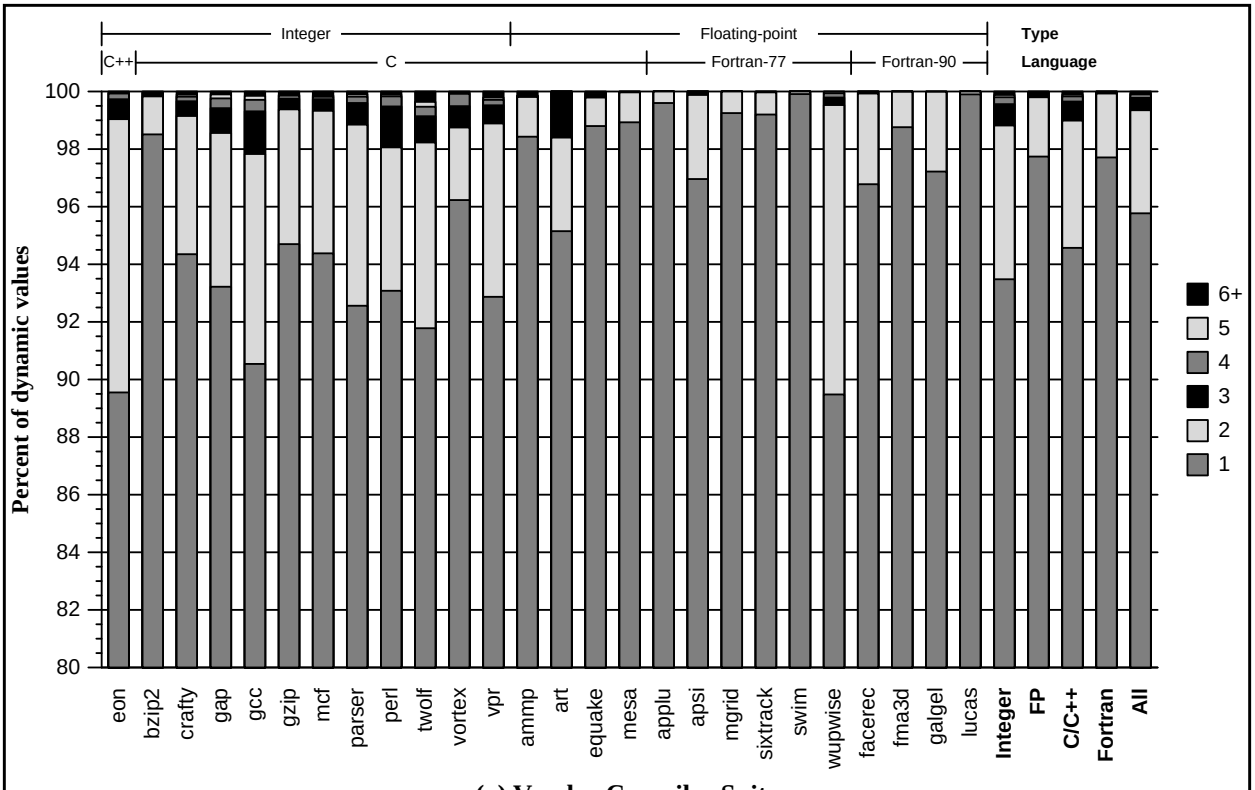
Figure 2.10 presents data on the temporal locality of the degrees of use arising from particular static instructions. The distribution shows how recently a value of the same degree of use came from the parent static instruction. In other words, if the last N unique degrees of use for each static instruction could be remembered, Figure 2.10 shows where the degree of use for the next value would occur in this set. The data indicate significant temporal locality: a value has the same as the degree of use as the last value from the same parent instruction 95% of the time; one of the last two observed degrees of use match the next one to occur over 99% of the time.

2.5 Mathematical Models

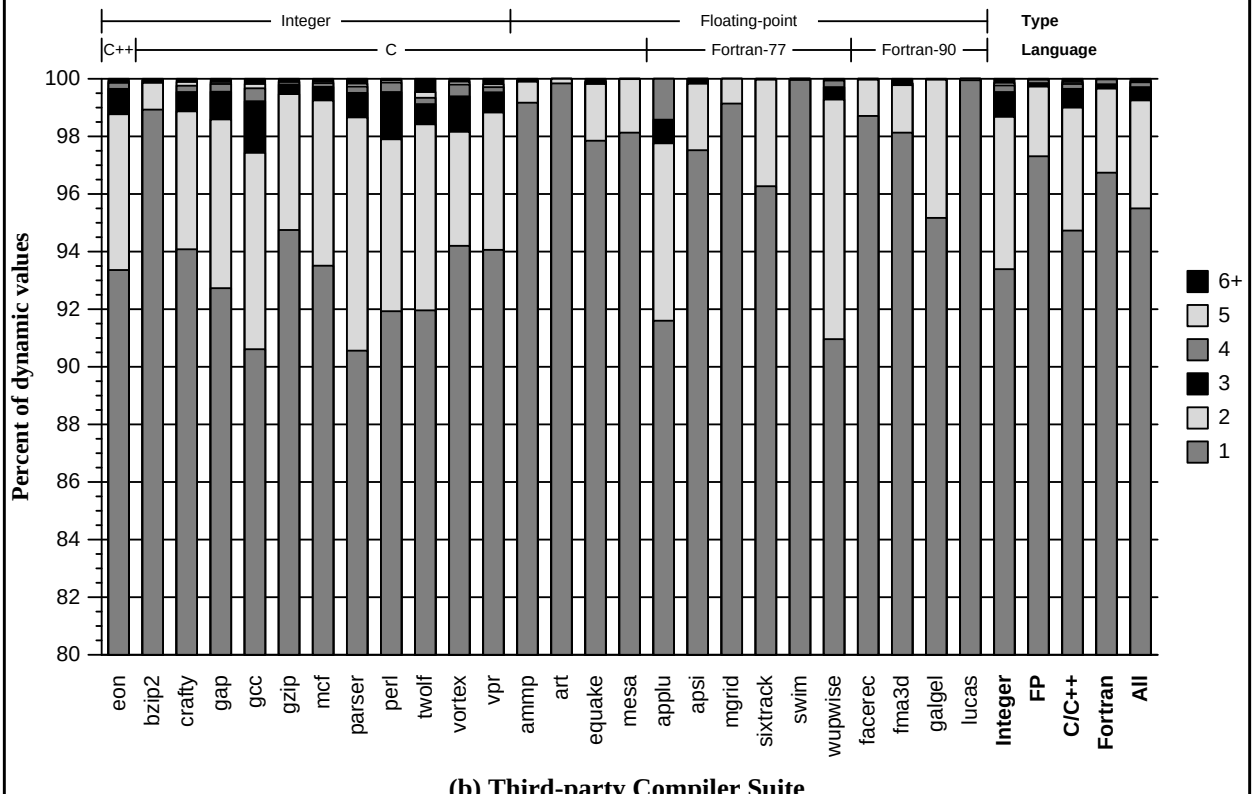
To this point, the properties of degree of use have been considered purely by experimental observation. Considering the properties of degree of use analytically can also be useful. One way in which an independent analytical model can be useful is to lend confidence to the correctness of experimental results. Another application is in answering questions for which experimental data is unavailable or difficult to obtain. This section presents mathematical descriptions of degree of use properties, improving a previously existing model and offering an independent confirmation of the observed mean degree of use.

2.5.1 Degree of use distribution

Eeckhout and Bosschere observed that the relative frequencies of values with different degrees of use (greater than zero) were well-fit by a power law model [28]. Mathematically, the probability that the degree of use D of a value is equal to x is given by:



(a) Vendor Compiler Suite



(b) Third-party Compiler Suite

Figure 2.10. Temporal locality in per-instruction degrees of use

$$P[D = x] = \alpha \cdot x^{-\beta}, \text{ for } x \in Z^{+} \qquad \text{(Eq. 1)}$$

where α and β were determined using linear regression. Taking the log of Equation 1 yields:

$$\log P[D = x] = \log \alpha - \beta \cdot \log x \qquad \text{(Eq. 2)}$$

which is linear in x. Note that by letting $x = 1$, it is easily shown that α is the frequency of single-use values.

Equation 1 cannot be a proper probability distribution function (PDF) for two reasons. First, it cannot account for the occurrence of values with a degree of use of zero, being undefined at that point. Second, a PDF must have the property that the probabilities of all possible values of the random variable sum to one:

$$\sum_{x = 0}^{\infty} P[D = x] = 1 \qquad \text{(Eq. 3)}$$

There is no such guarantee when α and β are empirically determined. However, it is possible to fix this model, simultaneously accounting for zero-use values and ensuring that the result is a true PDF.

Assuming that Equation 1 accurately describes the probabilities of all non-zero degrees of use, the fraction of values with degree of use zero is exactly the "leftover" probability when all non-zero degrees are summed:

$$P[D = 0] = 1 - \sum_{x = 1}^{\infty} P[D = x] = 1 - \sum_{x = 1}^{\infty} \alpha \cdot x^{-\beta} = 1 - \alpha \cdot \sum_{x = 1}^{\infty} \frac{1}{x^{\beta}} = 1 - \alpha \cdot \zeta(\beta) \qquad \text{(Eq. 4)}$$

where $\zeta(\beta)$ denotes the Riemann Zeta function.

Combining Equation 1 with Equation 4, a proper PDF is obtained:

$$P[D = x] = \begin{cases} 1 - \alpha \cdot \zeta(\beta) & x = 0 \\ \alpha \cdot x^{-\beta} & x \geq 1 \end{cases} \qquad \text{(Eq. 5)}$$

Note that this formula still has only two independent parameters, α and β, newly constrained by the requirement that $P[D = 0] \geq 0$. As a result, the distribution can be completely defined by the frequencies of zero-use and single-use values, although it remains to be seen whether the resulting distribution will match the empirical data as well as if the parameters were determined by fitting.

Table 2.4: Analytical Model Parameters

Method	α	β	<D>	$P[D = 0]$	$P[D = 1]$
Linear regression	0.832	2.79	1.58	–3.96%	83.2%
Mean and single-use	0.674	2.50	**1.75**	9.65%	**67.4%**
Mean and zero-use	0.717	2.55	**1.75**	**5.05%**	71.7%
Zero-use and single-use	0.674	2.35	2.34	**5.05%**	**67.4%**
Actual data	N/A	N/A	1.75	5.05%	67.4%

One nice side effect of having a PDF is that it allows for the analytical determination of the mean. The mean degree of use or expectation value $\langle D \rangle$ is simply the sum of all possible degrees of use weighted by their frequency of occurrence:

$$\langle D \rangle = \sum_{x=0}^{\infty} x \cdot P[D = x] = \sum_{x=1}^{\infty} x \cdot \alpha \cdot x^{-\beta} = \alpha \cdot \sum_{x=1}^{\infty} x^{-\beta+1} = \alpha \cdot \zeta(\beta - 1) \qquad \text{(Eq. 6)}$$

The above formula also offers additional methods for determining α and β without linear regression. Any two of (1) the probability of zero-use values, (2) the probability of single-use values, or (3) the mean can be manipulated to find the two PDF parameters without fitting. Table 2.4 shows the values of α and β derived by these methods and linear regression; the resulting mean and frequencies of zero- and single-use values are shown along with those from the actual data for comparison purposes. The bold values in each row were those used to determine α and β.

The requirement that the model be a proper PDF forces the selection of one of the methods that explicitly uses the frequency of zero-use values. Equation 5 ensures that when α and β are chosen to match the observed zero-use frequency, the aggregate probability will sum to one. Of the two possibilities, the one that also fits to the observed mean gives much better agreement to the free parameter. Note that linear regression results in quite poor agreement with both the mean degree of use and the frequency of single-use values. Also, justifying the initial criticism, linear regression does not yield a true PDF since the aggregate probability of all degrees of use greater than zero is 104% (witness the –4% leftover for zero-use values)! Neither of these problems exist with the improved model of Equation 6 with α and β derived from the observed mean degree of use and zero-use frequency.

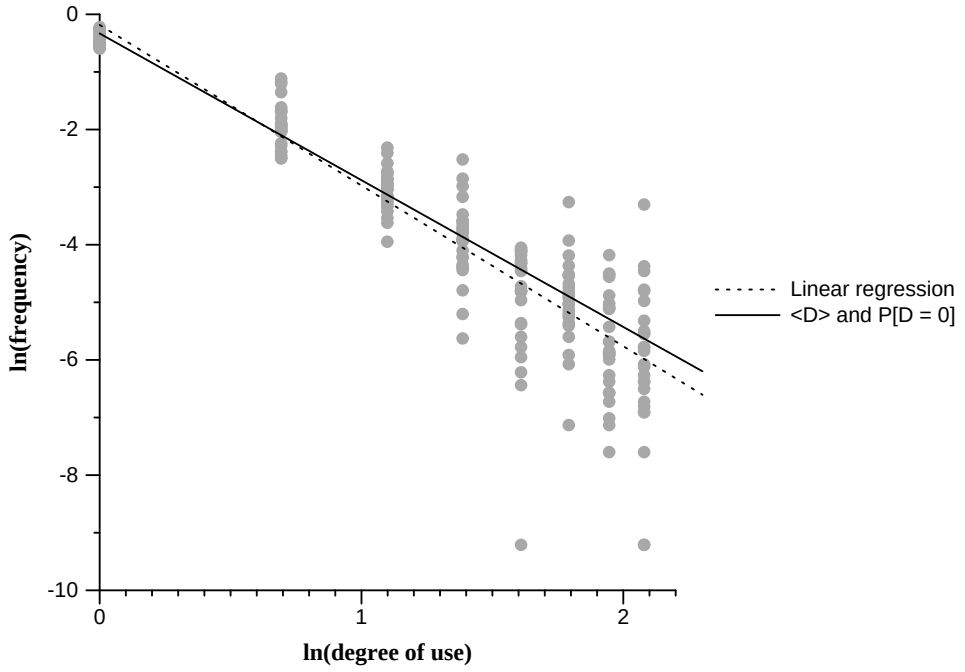


Figure 2.11. Analytical models of degree of use distribution

The fit of this model is shown graphically in Figure 2.11 along with the original linear-regression-based model. On this log-log plot, the degree of use-zero data cannot be included—the left-most datapoints are the frequencies of single-use values. Linear regression gives relatively more weight to the frequencies of high use values; Table 2.4 shows that as a result, it does not fit the data as well for either the most frequent degree of use (or the mean). The improved model does a much better job in this regard. Where the two models do deviate significantly (the frequencies of the highest-use values), the actual data shows far more variability, validating the improved model in this region.

2.5.2 Independent derivation of the mean degree of use

The preceding section provided probabilities for any possible degree of use and related them to the mean given a couple of parameters derived from observation. It is also possible to arrive at the mean degree of use by considering the properties of groups of instructions comprising a program.

The average degree of use $\langle D \rangle$ is just the total number of uses divided by the total number of values:

$$\langle D \rangle = \frac{N_{uses}}{N_{values}} = \frac{\langle u \rangle \times N_{insns}}{f_{vpi} \times N_{insns}} = \frac{\langle u \rangle}{f_{vpi}} \quad \text{(Eq. 7)}$$

where N_{uses} is the total number of uses, N_{values} is the total number of values, N_{insns} is the number of instructions, $\langle u \rangle$ is the average number of inputs per instruction, and f_{vpi} is the fraction of instructions that produce result values.

$\langle u \rangle$ and f_{vpi} are not independent, but may be estimated separately by considering a typical mix of dynamic instructions. Alpha machine instructions have between zero and three (inclusive) inputs. Zero input instructions are primarily used to load registers with constants; conditional moves are the only three input instructions (the Alpha architecture does not include fused multiply-adds). Neither category contributes more than 2% to the dynamic instruction count. Thus, $\langle u \rangle$ is essentially determined by the relative proportions of one- and two-input operations with loads and branches being the most important class of single-input instructions. Finally, an adjustment must be made for operations that take an immediate value as an input. Estimating f_{vpi} is more straightforward: as mentioned in Section 2.1, stores and branches comprise nearly all of the instructions that do not generate result values.

Starting with Equation 7 and applying these approximations,

$$\begin{aligned} \langle D \rangle &= \frac{\langle u \rangle}{f_{vpi}} \approx \frac{2 f_{2\text{-}input} + f_{1\text{-}input} - f_{immed}}{1 - f_{stores} - f_{branches}} \approx \frac{2(1 - f_{1\text{-}input}) + f_{1\text{-}input} - f_{immed}}{1 - f_{stores} - f_{branches}} \\ &= \frac{2 - f_{1\text{-}input} - f_{immed}}{1 - f_{stores} - f_{branches}} \approx \frac{2 - f_{loads} - f_{branches} - f_{immed}}{1 - f_{stores} - f_{branches}} \end{aligned} \quad \text{(Eq. 8)}$$

where f_{type} is the fraction of instructions of the given type. Using a mix of 17% branches, 9% stores, 26% loads, and 35% immediate mode operations (observed on DLX, an archetypical RISC architecture [38]), yields an average degree of use of 1.65, close to the observed means listed in Table 2.1. This result adds confidence to the notion that the mean degree of use will be consistent across different programs compiled to the same instruction set.

2.6 Summary

This chapter presented an exploration of inter-instruction value communication patterns via register degree of use. Several noteworthy properties were found to hold across a range of benchmarks, even when compiled with different compilers. Most importantly, single-use values

dominate all values produced during a program's execution, accounting for over 60% of values in most benchmarks; values with more than three uses account for under 10%. The high incidence of values with a small number of uses leads to an average degree of use of around 1.7, although this varies anywhere from 1.5 to 2.1 among the individual benchmarks. Together, these properties indicate that much of the value communication occurring during the execution of a program is simple in nature and thus should not require complicated mechanisms.

Two other interesting properties that could be exploited in the design of alternative communication mechanisms are the occurrence of zero-use values and the frequent use of high-use values as instruction inputs. The occurrence of zero-use values is highly-dependent on the specific benchmark and compiler, but can exceed 10%. These will be investigated in more detail in Chapter 4. While values with a high degree of use account for a small portion of all values produced, they supply a much larger fraction of all values used. Roughly one-third of instruction inputs come from values with more than seven uses (with another third from values with two to six uses and the remainder from single-use values). These *widely-used* values are precisely those that are well-suited to the register communication model in which a value is assigned dedicated long-term storage from where it can supply many consumers.

Examining degree of use behavior by architectural register and instruction type shows that a value's degree of use behavior is tied to its role within a program. For example, instructions that generate temporaries have a low average degree of use, while those that manipulate the stack pointer or other registers containing addresses generate more frequently-used values.

Because the purpose of each static instruction is fixed in a program, the per-instruction degree of use behavior shows little variation over the execution of a benchmark. More than 75% of all values come from static instructions that generate values with the same degree of use every time they are executed. Even among those instructions that can generate values with different degrees of use, the observed degrees of use are biased towards those generated most commonly and most recently. The per-instruction locality of degree of use is crucial to the success of degree of use prediction, which is presented next.

Chapter 3

Degree of Use Prediction

To guide the communication of a value, its degree of use must be known as soon as it is known that the value will exist, even prior to the computation of that value. However, degree of use information on a particular value cannot be obtained until the value is overwritten, after its communication has completed. The disparity between when degree of use information is needed and when it is known may be resolved through prediction and speculation.

High-performance microarchitecture speculate on many different kinds of information before the information may be observed or calculated. In each case, the speculation is enabled by the existence of a predictor for the information that is needed. For example, predicted outcomes of conditional branches are used by high-bandwidth fetch mechanisms before the branches have been executed. This chapter develops the concept of degree of use prediction, which will enable optimizations presented in subsequent chapters.

A feature common to all of the degree of use predictors described is the association of degree of use knowledge with the static instructions comprising the program. The structure of a program necessarily encodes the dataflow possible during any execution: the consumers of an instruction's result are only those instructions that use the result register and can be reached by that definition. Depending on the actual flow of execution, the same static instruction may give rise to instances that differ in the number (and identity) of consumers, but all of these possibilities are evident within the original program. This fact gives rise to the central role of the static instruction identity

in degree of use prediction and underlies the difference between static and dynamic degree of use prediction.

Static degree of use prediction involves analysis of a program to enumerate all possible degrees of use that can arise from each individual instruction. Perfect accuracy is attained for predictions on instances of instructions that only generate a single unique degree of use. Where the analysis finds that multiple degrees of use are possible, profile information or a predetermined policy can be applied to select the most likely or desirable prediction. In all cases, however, the prediction is associated with identity of the static instruction.

Dynamic degree of use prediction uses the observed behavior of the program during execution to predict its future behavior. Even in the dynamic scheme, the identity of the static instruction is of paramount importance. The use of dynamic prediction does not change the fact that the range of possible communication behaviors of each static instruction is fixed by the program. Therefore, a static instruction's identity is the best possible key with which to associate dynamic knowledge about instances of that instruction. Also, the *raison d'être* of a degree of use predictor is to supply information about a value before it is generated. As the processing of an instruction begins with a fetch operation on the instruction's address, that address (equivalent to the identity of the static instruction) is the first piece of information enabling the generation of a prediction.† The real potential of dynamic prediction schemes lies in their ability to leverage other information besides the identity of the executed instructions to differentiate among instances of a static instruction that behave differently.

This chapter begins with an explanation of how specific predictor implementations are evaluated. Next, the nature of the information provided by the degree of use predictor is discussed—in many cases, the exact degree of use may not be representable or even desired. A discussion of static degree of use prediction follows, including a complete description of how degree of use information for individual static instructions may be found through dataflow analysis. Most of the remainder of the chapter focuses on dynamic degree of use prediction, which uses past observations of the degree of use to generate subsequent predictions. Three different prediction algo-

---

† This statement is a simplification. In superscalar machines, for example, that fetch multiple instructions per cycle, only the starting address of a block of consecutive instructions may be explicitly generated for the fetch process. However, the argument is unchanged as the individual instruction addresses are trivially derivable from the block address.

rithms are presented, offering different trade-offs among performance, capacity, and complexity. Finally, the potential for hybrid prediction schemes, which combine elements of static and dynamic prediction schemes are addressed.

3.1 Predictor Evaluation

The prediction strategies in this chapter are presented without reference to any particular optimization. Not knowing how degree of use predictions are to be used, performance (or execution time) cannot be used to evaluate the efficacy of a predictor. Instead, indirect measures such as accuracy, that are independent of the application of the predictions, are used in this chapter. Unlike for a branch predictor, however, accuracy alone is insufficient to describe the complex behavior of a degree of use predictor. While a branch predictor supplies binary predictions for every branch, a general degree of use predictor supplies multi-valued predictions for a subset of value-producing instructions.

A degree of use predictor can be characterized by the relative numbers of overpredictions, underpredictions, non-predictions, and correct predictions (only three of which are independent). Overpredictions and underpredictions quantify predicted degrees of use greater than or less than the actual degree of use, respectively. The consequences of overpredictions and underpredictions may be vastly different depending on the application and the extent to which the prediction is incorrect. Absent an application, however, all mispredictions will be considered equivalent. Non-predictions count the number of values for which the predictor did not supply a prediction.

It is also useful to establish the notions of accuracy and coverage. *Accuracy* is defined in the typical fashion—the percentage of all predictions that are correct. Non-predictions do not influence accuracy but are instead reflected in *coverage*, which is defined as the percentage of all values for which a prediction (right or wrong) is generated. Figure 3.1 defines these relationships graphically. Note that increasing coverage alone increases correct predictions *and* mispredictions, while increasing accuracy alone changes mispredictions into correct predictions. Provided the accuracy is above a minimum threshold (established by the benefit of correct predictions versus the cost of mispredictions), increasing either accuracy or coverage in isolation results in a net benefit.

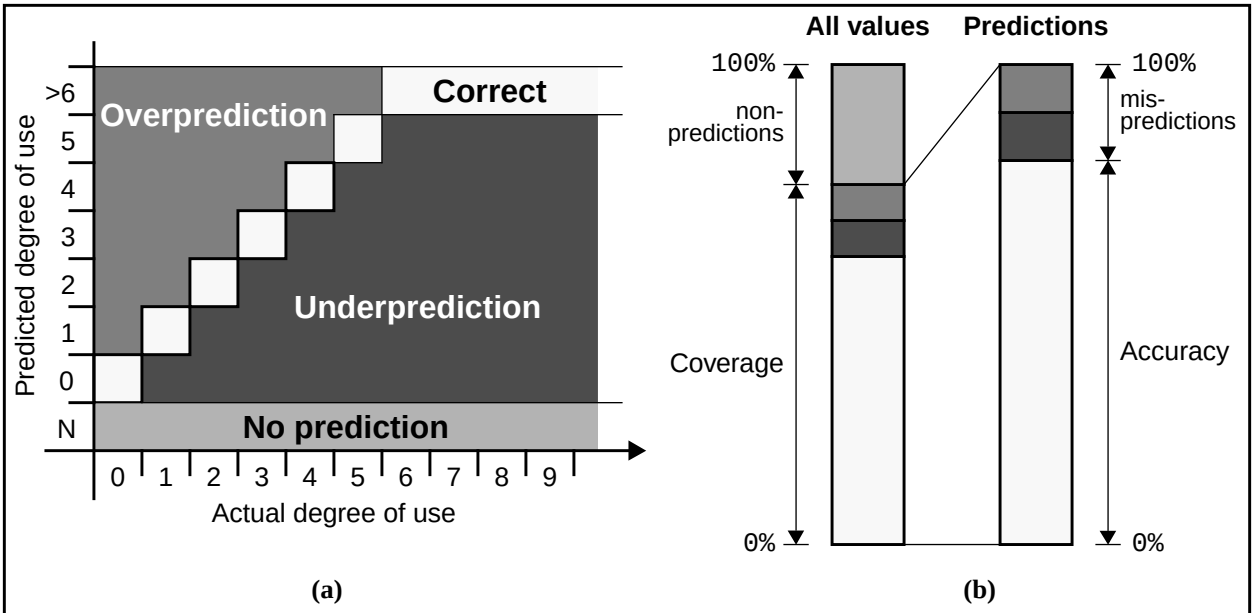


Figure 3.1. Accuracy and coverage in degree of use prediction

Many predictor parameters offer a trade-off between accuracy and coverage. The optimal value of any such parameter ultimately depends on the application, but it is possible to make some observations using a simple model. Assume that any performance benefit made possible by degree of use prediction occurs in proportion to the number of correct predictions. This benefit is offset by a performance cost in proportion to the number of mispredictions (a simplification, since mispredictions occur in different magnitudes and directions). By expressing the average cost of a misprediction as a multiple (the *cost factor*) of the average benefit of a correct prediction, an effective performance benefit can be calculated:

$$B = n_c - f \cdot n_m = N \cdot c \cdot a - f \cdot N \cdot c \cdot (1 - a) = N \cdot c \cdot [a - f \cdot (1 - a)] \qquad \text{(Eq. 9)}$$

where B is the effective benefit, n_c is the number of correct predictions, n_m is the number of mispredictions, N is the total number of instructions, c is the coverage, a is the accuracy, and f is the cost factor. Note that the magnitude of the benefit is directly proportional to the coverage, which is good as long as the benefit is positive.

Achieving a positive effective benefit requires the factor in brackets to be positive, which establishes a constraint on the minimum accuracy:

$$a > \frac{f}{f + 1} \qquad \text{(Eq. 10)}$$

High cost factors (i.e., more costly mispredictions) demand a more accurate predictor, and make it more difficult to justify sacrificing accuracy for coverage. For example, given 80% coverage and 90% accuracy, a decrease in accuracy to just 89% must be met by a coverage increase to better than 94% to achieve a net performance improvement at a cost factor of five. Even at a cost factor of one, the 1% accuracy decrease is only offset by a 2% coverage increase. Thus, within the constraints of complexity and hardware cost, it will nearly always be better to choose policies that lead to more selective predictors (i.e., those that increase accuracy at the expense of coverage).

3.2 Encoding Degree of Use Information

Before exploring how degree of use prediction may be accomplished, the form of the prediction itself will be considered. This discussion is meant to provide an overview of the possibilities as the specifics will depend on how the predictions are to be used. As applications are discussed in the following chapters, this topic will be revisited with the specific needs of the applications in mind.

Figure 3.1 implies certain characteristics about the information available from the predictor. Specifically, there is a limit (six uses in the example) beyond which all degrees of use are considered equivalent from the predictor's point of view. Below this limit, the predictor differentiates each possible degree of use. Finally, the degree of use predictor may abstain from generating a prediction at all. This section discusses the issues surrounding predictor policies regarding the maximum degree of use, the preference for certain outcomes, the grouping together of different degrees of use, and the choice of a default behavior.

3.2.1 Maximum predictable degree of use

The huge range of potential degrees of use of dynamic values (see Section 2.1) must be considered in the design of a degree of use predictor. Values with very high degrees of use occur infrequently relative to other values. While it may be important to identify high-use values, it almost certainly does not matter whether such a value has a degree of use of one hundred thousand or one hundred million—in both cases, the live time of the value is very large compared to the lifetime of a typical instruction within the processor, and the value will be needed by many consumers long after the generating instruction retires. Also, there is the practical issue of physically representing

degree of use information. Allocating tens of bits to a degree of use prediction when three bits suffices over 99% of the time is clearly wasteful.

These considerations lead to the selection of a degree of use limit. Thus, all degree of use predictors are saturating: degrees of use greater than the limit are treated as equivalent to the limiting degree of use. The precise limit is dictated by the representation and storage overhead, the diminishing ability of a predictor to distinguish among high degrees of use (see Figure 3.20), and the particular application. Beneficial side effects of reducing the maximum predictable degree of use include increased accuracy and a potential decrease in predictor overhead (if the degree of use can be represented with fewer bits).

The choice of encoding of the degree of use information leads to a spectrum of different possibilities. An example two-bit encoding might differentiate instructions that: (1) generate single-use values, (2) generate values with some other single degree of use, (3) generate values with many uses, (4) have variable/indeterminate behavior. The final choice of encoding necessarily would depend on the number of available bits and the expected application of the information. Considering that a majority of values have one of a small number of degrees of use (Figure 2.1), most interesting applications could probably be handled with two or three bits of degree of use information per instruction. For the predictors studied in this chapter, a three-bit encoding is assumed, allowing degrees of use less than seven to be fully differentiated.

3.2.2 Biasing

Biasing refers to the practice of preferentially selecting certain possible degrees of use based on the application. It is frequently possible for instructions that are indistinguishable to the predictor to have different degrees of use. For a static predictor, this manifests as more than one degree of use being possible for a given instruction with no additional information favoring a particular outcome. The same situation applies to a dynamic predictor when identical input information corresponds to different possible degree of use outcomes. The application of the degree of use prediction may indicate that, for example, the highest possible or observed degree of use be delivered under these circumstances. In this case, the predictor would be biased towards the maximum degree of use. Other biases that might be reasonably expected to be useful are biasing towards the minimum degree of use, the most likely degree of use (exactly the same as minimum with the

exception of degree of use zero), and (for dynamic predictors only) the most recently observed degree of use. Biasing is not employed by any of the predictors that will be presented.

3.2.3 Default predictions

Any application of degree of use prediction must be able to handle the unavailability of a prediction for certain instructions. This situation applies to both dynamic and static prediction methods. Dynamic schemes may lack information on instructions before they are first executed or if a long time has elapsed since their last execution; static degree of use information may not be available within certain dynamically-linked libraries or may be deliberately omitted for instructions that have statically-indeterminate behavior. In such cases, a default degree of use prediction can be supplied by the predictor in place of a non-prediction.

The use of such an implicit default prediction can reduce predictor overhead. In order to handle non-predictions, any application of degree of use information will have an implicit behavior in the absence of information, which may match that corresponding to a specific degree of use prediction. In this case, explicit demarcation or storage within the predictor of the degree(s) of use leading to that default behavior is unnecessary as predictions of this degree are correctly subsumed by the default prediction. The savings correspond to the frequency of occurrence of the default degree of use; the cost of using a default prediction is the loss of information about the confidence of a particular default prediction.

Because different applications of degree of use prediction will be studied and the default is an application-dependent policy, implicit default predictions are not considered for the evaluation of the predictor in isolation. Instead, the predictor is allowed to deliver a non-prediction as a distinct outcome from any particular degree of use.

3.2.4 Grouping

Grouping multiple degrees of use into classes (e.g., many-use or few-use) is another potentially beneficial predictor policy. For certain applications, knowing that the degree of use of a value lies within a certain range is more important than knowledge of the exact degree of use. Predicting that the degree of use of a value will be in a specific range is easier than attempting to predict the exact degree of use because the latter can exhibit mispredictions due to confusion between degrees of use belonging to a single group. Thus, a predictor that employs grouping will always

have higher accuracy. Note that the saturating maximum degree of use (Section 3.2.1) is equivalent to a grouping of all predictions greater than or equal to that limit. In this chapter, absent a particular application, it will be assumed that it is important to differentiate among all the degree of use outcomes below the limit.

3.3 Static Degree of Use Prediction

This section develops static degree of use prediction. The defining characteristic of static degree of use prediction is the generation of predictions through off-line analysis. Such schemes rely upon the compiler or a profiler to annotate each value-generating static instruction with degree of use information. This information can then be conveyed to the hardware to generate degree of use predictions.

The capabilities of a static prediction scheme depend on the sophistication of the analysis performed and the expressiveness of the interface used to communicate analysis results to the hardware. For example, it may be possible to communicate multiple possible degrees of use per instruction along with the dynamic conditions that lead to one particular outcome. In this section, however, it is assumed that the goal of the static analysis is the assignment of a single degree of use to each static instruction.

The characterization data presented in Section 2.4 illustrates the potential for obtaining reasonable performance from such a static prediction scheme. Specifically, the majority of static instructions generate values that have only a single unique degree of use during a program's execution; even among those generating values with different degrees of use, one particular degree of use dominates. However, this data applies to one specific execution. Since Figure 2.6 shows that an average of 40% of static instructions have more than one statically-possible degree of use, one must ask the question of how well a single, statically-selected degree of use for such an instruction would suffice across executions.

Executions differ only as a result of input data—the program itself does not change. Therefore, any variability in the degrees of use must result from differences in the input data. Since the degree of use of a particular dynamic instruction is completely determined by the program (fixed) and the subsequent dynamic control flow (variable), this issue is equivalent to a more familiar one—the effect of input data on dynamic control flow.

Others have observed that the control flow of a program is relatively constant with respect to the input data [33, 86]. Also, consider the categorization of static instructions into those with: (1) a single static degree of use; (2) a single degree of use where the analysis derives multiple possible degrees of use (e.g., because certain paths are impossible due to the logic of the program); (3) two possible degrees of use where one only occurs in the presence of a rare error condition; (4) multiple possible degrees of use where many inputs lead to the same single degree of use; and, (5) highly-input dependent multiple degrees of use. Only those static instructions in the final category will contribute significantly to variability in degree of use characteristics across different executions. Therefore, it can be expected that the performance of a static prediction scheme with a fixed, single degree of use per static instruction will be robust with respect to varying input data.

The actual process of static determination of degree of use information involves dataflow analysis similar to that already performed by optimizing compilers. The efficacy of this technique alone is limited because it computes for each instruction every possible degree of use, including those that result from impossible or unlikely paths through the program. Without additional information, no prediction may be safely selected where the analysis indicates multiple possible degrees of use, resulting in limited coverage (but 100% accuracy!). The coverage can be improved with varying accuracy degradation using profiling: data from branch or path profiles used during the dataflow analysis can identify the most likely prediction among the set of possible predictions. The direct application of degree of use profiles to improve the results of the dataflow analysis is considered in Section 3.3.4. Augmenting the analysis itself with control-flow profiling information is also possible [7, 21, 62, 70], but is outside the scope of this work.

3.3.1 Formulating degree of use determination as a dataflow problem

A dataflow problem is simply a system of equations associated with a control-flow graph whose solution yields information about the data in the program represented by the graph. The variables within the system of equations are the dataflow *facts*, one at each node. The *direction* of the problem—forward or backward—determines whether dataflow information propagates in the same or opposite direction as the flow of execution, respectively. A *meet operator* specifies how multiple facts are combined into a single fact. Finally, each node in the graph has an associated *dataflow function*, which summarizes the dataflow effect of that node, or how that node changes a dataflow

fact. Thus, to define the degree of use dataflow problem, each of (1) the kind of dataflow facts, (2) the direction of the problem, (3) the meet operator, and (4) the possible dataflow equations must be specified.

At each point in the program, the dataflow fact for a register R is the set of all possible numbers of uses of R between that point and the end of the program. When the point under consideration is the instruction that writes R, the dataflow fact for R is exactly the set of possible degrees of use for that instruction, which is precisely the information required by a static prediction scheme. A maximum representable degree of use D_{max} must be defined to avoid infinite-sized facts.† Thus, for any given register, the facts are represented by a set S, where $\forall\, u \in S$, $0 \leq u \leq D_{max}$.

Note that the dataflow problems for the architectural registers are completely independent: the dataflow facts (i.e., possible degrees of use) for one register never affect the facts about another register. Thus, without loss of generality, the discussion of the dataflow problem can be simplified by limiting it to a single architectural register. The overall solution consists of the set of independent solutions for all architectural registers, which may be determined in parallel.

The determination of degree of use is a backward dataflow problem since the facts at each point pertain to paths *from* that point to the exit. Each instruction that writes a register defines a new value. For uses of that value to be attributed to the instruction, information must flow from the uses and be collected at the definition (i.e., backwards with respect to execution). Thus, each definition (1) assumes the dataflow facts true immediately after the definition as the possible degrees of use for its values, and (2) creates a new fact {0} for the overwritten register (true immediately prior to the instruction) that indicates that once the flow of control reaches this definition, no more uses of the prior contents of the register can occur.

Consider next the meet operation to combine dataflow facts along two potential paths. If different sets of degrees of use are possible along two different paths from a certain point, then the set of possible degrees of use prior to that point includes the elements of both sets. Given facts U and V (representing sets of possible degrees of use for a particular register), true for two different

---

† For example, consider a value used within a loop body. Since loop iteration counts are opaque to the analysis, the fact for the register containing that value must account for all possible iteration counts $[1, \infty)$. Without an upper bound, the size of the dataflow fact would be infinite. A further discussion of the maximum degree of use may be found in Section 3.2.1.

potential paths from a point, the facts true at that point are U ∪ V. Thus, the meet operator for this dataflow problem is set union.

Finally, consider the dataflow functions for an instruction, which modify the facts to account for the inclusion of that instruction on the path. Any instruction may be decomposed into uses of some registers (occurring first) and (sometimes) the definition of a register. Because degree of use determination is a backward dataflow problem, the dataflow function must convert the facts true after the instruction is executed to those that are true beforehand. Therefore, the register definition modifies the facts first: if the instruction writes the register R, the set of possible degrees of use corresponding to that register is made to contain only the zero element (not the empty set: it is *known* immediately prior to the definition that R will be used exactly zero times prior to it being overwritten). Subsequently, the facts are modified by the input registers of the instruction. For each input register, every element in the set of possible uses corresponding to that register is incremented. For example, if the set of possible uses of a register after an instruction is {2, 5, 6}, then a single use of that register by the instruction leads to the set {3, 6, 7} applying prior to the instruction. The dataflow functions are as follows:

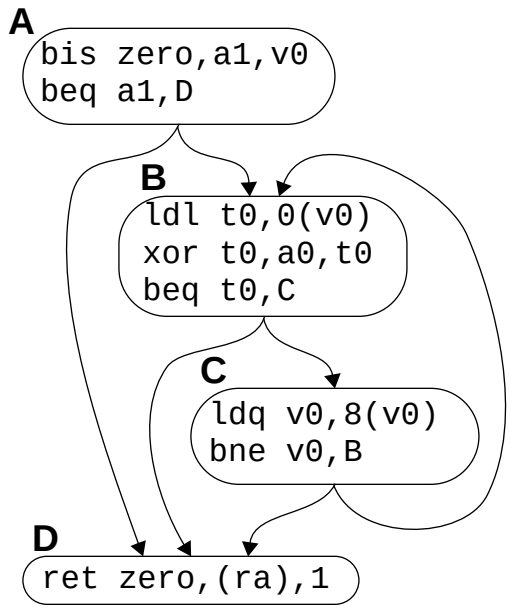
$$\begin{array}{ll} \text{Definition of R:} & \lambda S.\{0\} \\ \text{Use of R:} & \lambda S.\{min(u+1, D_{max}) \,|\, u \in S\} \end{array} \qquad \text{(Eq. 11)}$$

where S is the set of dataflow facts corresponding to the register R.

An example instance of the dataflow problem and its solution appears in Figure 3.2. The code describes a simple function that finds the first node within a linked list that contains the specified data. The control-flow graph for the `find()` function appears along with the dataflow facts that true at the entry and exit of each basic block. The derivation of facts true at points within a basic block is trivial given the correct facts at the exit of the block.

In order to proceed with the dataflow analysis, initial facts true at the procedure exit (`return` instruction) must be provided from which the dataflow information for the rest of the `find()` procedure may be derived. Because architectural registers do not observe procedural boundaries, the uses of values past the end of a procedure are actually determined during analysis of the calling procedure. The need to initialize the facts at a procedure's exit based on subsequent uses in the caller illustrates the need for interprocedural dataflow analysis, wherein dataflow facts can be propagated among the procedures of the program. The general approach used for the interproce-

```
node_t *find(int val, node_t *list) {
   node_t *node = list;
   while (node && (node->data != val))
      node = node->next;
   return node;
}
```



	v0 (node)	**a0** (val)	**a1** (list)	**ra**
A.enter	{0}	{0,1,2,3,...}	{2}	{1}
A.exit	{1,2}	{0,1,2,3,...}	{0}	{1}
B.enter	{2}	{1,2,3,...}	{0}	{1}
B.exit	{1}	{0,1,2,3,...}	{0}	{1}
C.enter	{1}	{0,1,2,3,...}	{0}	{1}
C.exit	{1,2}	{0,1,2,3,...}	{0}	{1}
D.enter	{1}	{0}	{0}	{1}
D.exit	{1}	{0}	{0}	{0}

Figure 3.2. Control flow graph annotated with degree of use dataflow facts

dural analysis was described by Sharir and Pneuli [75] and is detailed in Section A.3 of the appendix. In Figure 3.2, it is assumed that only the return value is used by the calling function (and only once).

3.3.2 Solving the degree of use dataflow problem

Now that the dataflow problem has been completely specified, consider the following method of solving it. Enumerate all paths through the program. For each such path, solve the dataflow problem assuming the execution traverses only that path. Finally, combine the resulting solutions using the meet operator (since the solutions are sets of facts). The resulting solution is called the meet-over-all-paths solution and is the most precise possible solution assuming that all paths through the program are actually feasible. Obviously, this method of solution is intractable for anything but the most trivial programs.

Instead, the dataflow problem may be solved iteratively as follows. First, all of the nodes in the control-flow graph are placed in a queue. In each step, a node is removed from the queue and the facts true after that node are computed by combining (using the meet operator) the facts true before each of that node's successors. Then, the node's dataflow function is applied to find the before fact for the node. If the before fact is changed, the node's predecessors are placed in the queue. Assuming that the process terminates (i.e., the facts converge), the resulting solution is called the greatest fixed-point solution. The nature of the particular dataflow facts and functions of the degree of use problem make it possible to provide guarantees about the existence and precision of a solution arrived at by this method [49].

First, the domain of the dataflow facts is a complete lattice—that is, it is a finite, partially-ordered set (ordered by $\supseteq$) with a least upper bound $\varnothing$ and a greatest lower bound $\{u \mid u \in \mathrm{Z}, 0 \leq u \leq D_{max}\}$. Second, the dataflow functions of Equation 11 are monotonic: $\mathrm{P} \supseteq \mathrm{Q} \Rightarrow \mathrm{f(P)} \supseteq \mathrm{f(Q)}$.† Together, these properties guarantee the existence of a greatest fixed-point solution to the set of dataflow equations. Additionally, because the dataflow functions are distributive under the meet operator: $\mathrm{f(P \cup Q)} \equiv \mathrm{f(P)} \cup \mathrm{f(Q)}$,‡ this solution is guaranteed to be the same as the meet-over-all-paths solution.

3.3.3 Results

The dataflow analysis described in the previous section was performed using a binary analyzer. Normally, this type of analysis would be performed by a compiler, but modifying the compilers used in generating the evaluation benchmarks was not an option. The consequences of performing the analysis directly on a binary and the operation of the analyzer are described in Section A.3 of the appendix.

Dataflow analysis yields the set of possible degrees of use for each static instruction of the program. Because the analysis is safe, it will never be the case that a degree of use occurs that is not identified by the analysis process. Therefore, a single, static, perfectly-accurate prediction may be assigned to each static instruction for which the analysis finds only one possible degree of

† For the definition dataflow function, $\{0\} \supseteq \{0\}$ regardless of P and Q. For the use dataflow function, $\mathrm{P} \supseteq \mathrm{Q} \Rightarrow \mathrm{P} \equiv \mathrm{P} \cup \mathrm{Q} \therefore \mathrm{f(P)} \equiv \mathrm{f(P \cup Q)} \equiv \mathrm{f(P)} \cup \mathrm{f(Q)} \supseteq \mathrm{f(Q)}$. This proof assumes distributivity under set union $\mathrm{f(P \cup Q)} \equiv \mathrm{f(P)} \cup \mathrm{f(Q)}$, which is proved below.

‡ The proof is trivial for the definition dataflow function which always yields $\{0\}$. For the use dataflow function, $\mathrm{f(P \cup Q)} \equiv \{\mathrm{f(u)} \mid \mathrm{u} \in (\mathrm{P} \cup \mathrm{Q})\} \equiv \{\mathrm{f(u)} \mid \mathrm{u} \in \mathrm{P}\} \cup \{\mathrm{f(u)} \mid \mathrm{u} \in \mathrm{Q}\} \equiv \mathrm{f(P)} \cup \mathrm{f(Q)}$.

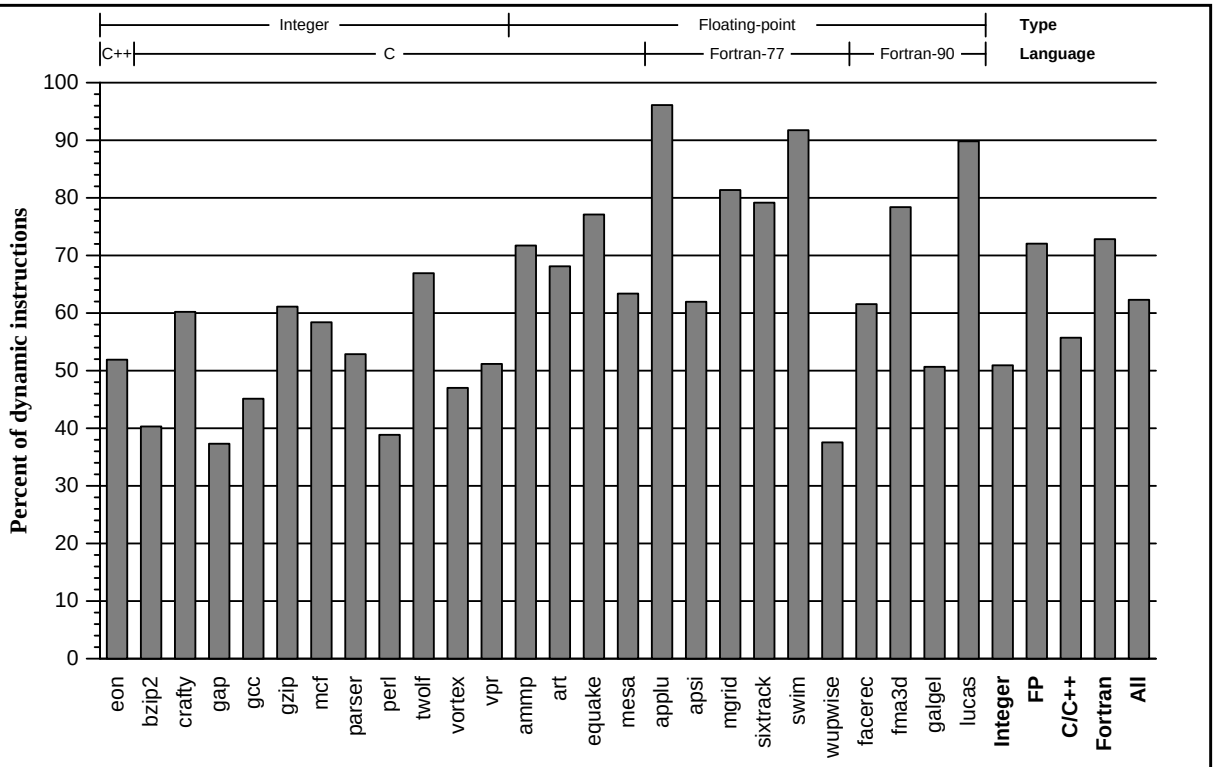


Figure 3.3. Static prediction using dataflow analysis

use. If these are the only predictions made, the aggregate accuracy will be 100% and the coverage will equal the percentage of dynamic instructions receiving a correct prediction; these results appear in Figure 3.3. The coverage spans a large range from 37% to just over 96%, but averages an impressive 62%. The floating-point benchmarks, which in general exhibit less complicated control-flow, have a higher average portion of static instructions with a single, statically-identifiable degree of use.

3.3.4 Applying profile information

Improving the capability of static degree of use prediction any further requires handling instructions for which the analysis identifies multiple possible degrees of use. While some of these instructions certainly generate dynamic instances with different degrees of use, the difference between Figure 2.5 and Figure 2.6 shows that some static instructions generating single degrees of use are simply not identified by the static analysis. Such instructions are of one of two types: (1) instructions constrained by the program to *always* generate a single degree of use but not recognized as such due to the limitations of the analysis, and (2) instructions for which multiple

degrees of use are actually possible, but do not occur in some executions. Regardless of the type, additional information is required to select a suitable prediction from those identified by the static analysis, and this task requires profile information. Control-flow (e.g., branch or path) profiles used in combination with a more sophisticated dataflow analysis algorithm could provide this information, but degree of use profiles (generated by execution-driven simulation) supply this information directly.

Comparing Figure 3.3 with Figure 2.7, there is clearly room to improve the static coverage while maintaining perfect accuracy. Consider first using the same program inputs during profiling and evaluation. Making additional predictions (beyond where the analysis indicates a single degree of use) only for those instructions with a unique degree of use in the profile gives the highest possible coverage attainable at perfect accuracy (equal to the size of the lowermost bars in Figure 2.7). The profile also reflects single-degree-of-use-instructions that *were* identified as such by the static analysis, begging the question of why the static analysis is performed at all. The answer is that in reality, the predictions will be applied to a program run with different inputs than used to generate the profile. Therefore, priority must be given to any information determined statically; how the (more or less) accurate profile data is applied involves a trade-off between accuracy and coverage.

After annotating those instructions that yield to the dataflow analysis (i.e., can be proven to always generate the same degree of use), one is left with a set of static instructions without a prediction. The application of the profiling data involves choosing, for each such instruction, what, if any, prediction to assign. The selection of a prediction for a given instruction is trivial: choosing whatever degree of use occurred most frequently for that instruction must (modulo the accuracy of the profile) result in the highest predictor accuracy.† Choosing which instructions receive predictions is more difficult (excluding the trivial case in which the profile does not include the instruction—any prediction made in this case would be no better than a guess made without the benefit of a profile). Each additional instruction assigned a prediction increases the coverage, but some will reduce the accuracy substantially.

---

† The final measure is, as always, performance. Some policies that give lower accuracy as defined here may afford better performance in certain applications. Biasing, discussed in Section 3.2.2, is one example of such a policy.

The most conservative policy will attempt to identify those instructions that only generate a single degree of use, but that were not identified by the static analysis. All such instructions will exhibit a single degree of use in the profile itself (with 100% frequency of occurrence). The least conservative policy assigns a prediction for every instruction for which a single outcome occurs more often than all others combined (i.e., has greater than 50% frequency of occurrence of the highest degree of use). In between these two extremes is a continuous spectrum of policies characterized by a frequency threshold. The threshold is the minimum frequency of occurrence of the dominant degree of use required for that degree of use to be assigned as a prediction for an instruction. Lowering the threshold increases coverage at the expense of accuracy.

Figure 3.4 shows how accuracy and coverage vary with this threshold. Each gray line represents a single benchmark while the darker line indicates the average. The profile data were generated using the test inputs while the prediction was performed using the usual train inputs (see Section A.1.1 of the appendix for details on the benchmark inputs).

The values of the accuracy and coverage indicate the quality of the profile data. For example, three benchmarks exhibit relatively low coverage that is constant with threshold. In these cases, many instructions left after the static analysis (i.e., those that have multiple possible degrees of use) were absent from the profile data also, limiting the coverage attainable regardless of thresh-

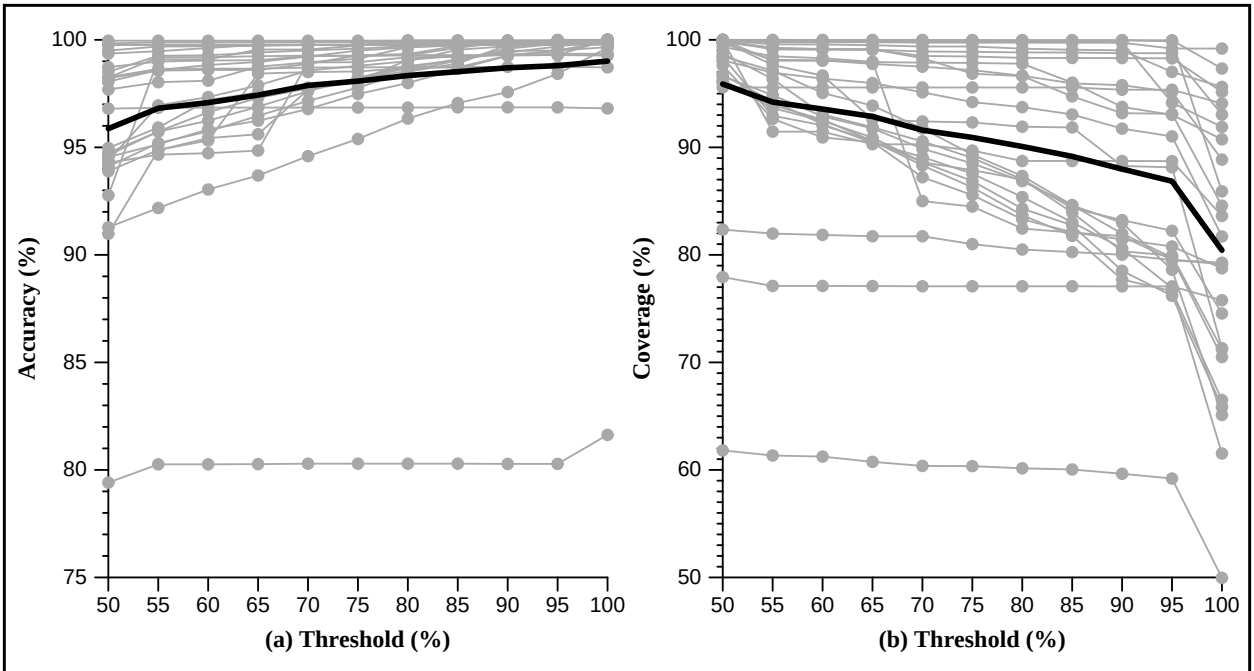


Figure 3.4. Static predictions derived from degree of use profiling

old. In one of these cases, the accuracy was also low, indicating that even those instructions that were in the profile behaved much differently between the two runs. Another interesting feature of the data is the relatively large jump in coverage when reducing the threshold from 100% to 95%, implying that many values originate from instructions that can generate more than one degree of use, but are highly biased. That the accuracy does not show the same behavior indicates that the dominant degree of use for many of these instructions is the same for both inputs (the profile and the evaluation).

Figure 3.5 portrays benefit (see Section 3.1) as gradations in shading (with darker shades indicating more benefit) versus threshold and cost factor. The shading range is normalized at each cost factor (i.e., the same shade at different cost factors does not represent the same absolute benefit). A contour line illustrates the threshold for each cost factor where maximum benefit is obtained. As the cost factor rises, accuracy becomes relatively more important and the optimal threshold value increases. Even for very high cost factors (>15), however, it is still beneficial to have a threshold less than 100%. The vertical extent of the shaded area at each cost factor indicates the sensitivity of the performance to the threshold. At low cost factors, the performance is relatively insensitive to the threshold, but as misprediction costs become more dominant, perfor-

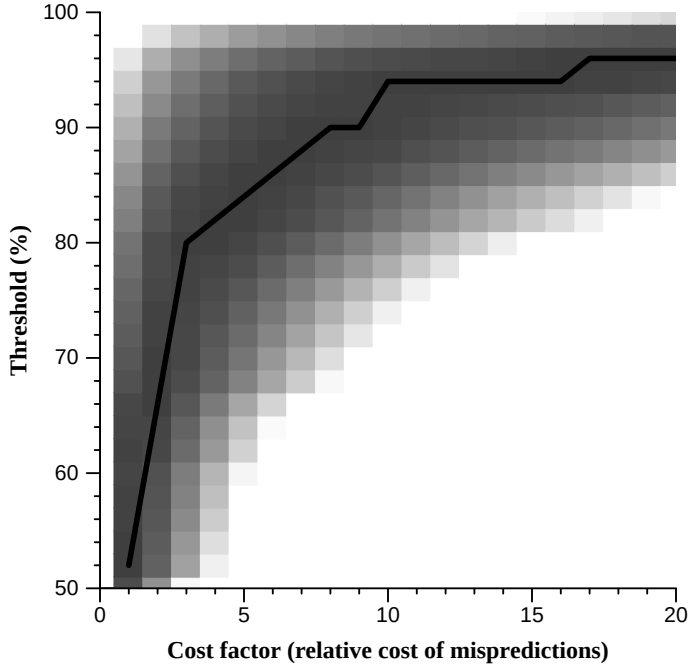


Figure 3.5. Optimum threshold value versus misprediction penalty

mance decreases more rapidly as threshold deviates from the optimal value. Therefore, this model indicates that a threshold of around 95% is a reasonable choice for a range of applications.

3.3.5 Communicating static predictions to the hardware

However obtained, to be useful in runtime communication optimization, statically-derived degree of use information must be somehow passed to the hardware. Details on the various techniques for communicating information to the runtime are outside the scope of this thesis, although some possibilities that assume the ability to modify the instruction set are presented here.

Given complete freedom in the design of the instruction set, degree of use information can be included directly in the instruction encodings. As degree of use information applies only to instructions that produce a result, instructions with a destination register field are simply augmented with an additional field for degree of use information. Alternatively, certain architectural registers could be dedicated to different degree of use classes (e.g., single-use or few-use temporaries or long-lived many-consumer). Such methods could only be used in a new design where the architectural interface is not yet fixed. One might also imagine more evolutionary changes, involving, for example, new instruction prefixes or unused instruction encodings.

One important issue with architecting the encoding of degree of use information is its immutability. Applications will generally need different kinds of degree of use information: some may only need to know about single-use values while others may require more information. The two applications presented in this dissertation illustrate this quite well: useless instruction elimination (Chapter 4) only requires the identification of a certain class of values, while use-based register caching (Chapter 5) depends upon knowing the exact degree of use for every value. If the architected interface is not sufficiently generic, some optimizations may not benefit from a particular static encoding.

3.4 Dynamic Degree of Use Prediction

A dynamic degree of use predictor uses the observed run-time behavior of a program to generate its predictions. While static degree of use prediction can offer superb accuracy, it has three notable shortcomings. First, the coverage is limited by the precision of the analysis and the availability of good profile information. Second, the availability of static predictions for a program of

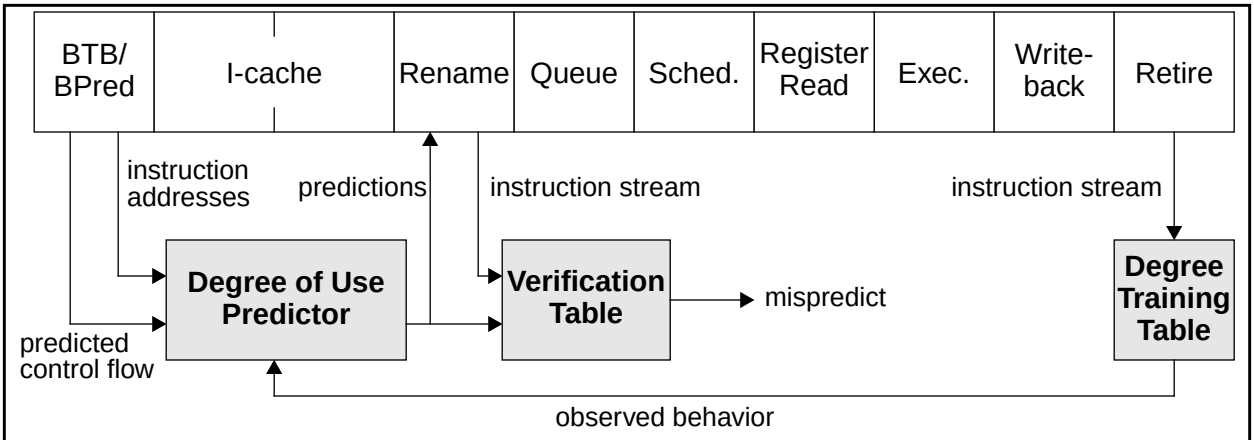


Figure 3.6. A dynamic degree of use predictor in a processor pipeline

interest depends on *a priori* analysis.† Finally, as just discussed in Section 3.3.5, an interface must exist to communicate the predictions to the hardware, which may constrain the amount and format of information that can be communicated to the implementation. Dynamic prediction suffers none of these problems as it profiles the actual program of interest at run time from within the implementation.

An overview of how a degree of use predictor might interface with a prototypical processor pipeline is shown in Figure 3.6. A storage structure maintains per-instruction state used to generate predictions. It is accessed with instruction addresses from the front end in parallel with the fetch of the instructions from the instruction cache. Additional information (e.g., control flow predictions) may be used in generating the final prediction, which is available by the time the instruction's registers are renamed. Components for training the predictor and verifying the predictions are also needed. These structures observe the uses and definitions in the instruction stream to calculate actual degrees of use. The observations may be performed anywhere within the processor pipeline, although different locations offer different trade-offs among complexity and performance. Although both training and misprediction detection require observation of the instruction stream, there is no requirement that the same instruction stream be used in both cases.

All of the dynamic predictor designs presented in this section operate in two separate steps. First, all or part of the address of an instruction is used to access some per-instruction information.

---

† A run-time system could conceivably be used to perform on-line dataflow analysis, but its effort would be spent much more profitably generating predictions based on direct degree of use profiling—in which case system is really performing dynamic prediction.

Second, the state information is used (perhaps with other external information) to generate the final prediction. The first step determines only the potential availability of a prediction for a particular instruction (i.e., the predictor's maximum coverage). The predictor's accuracy, however, is determined entirely by the algorithm used to turn the per-instruction state into a prediction. A prediction may not be generated even in cases where an entry exists for an instruction. Thus, the prediction algorithm can cause the coverage to be lower than the simple availability of per-instruction state, but it cannot increase it beyond this level. This division of the task of degree of use prediction simplifies the exploration of the large predictor design space.

The next sections present the various predictor algorithms beginning with the simple strategy of returning the last observed degree of use for a each static instruction. Then, a confidence mechanism is added, significantly improving the accuracy. Finally, the use of control-flow information enables predictors that distinguish among multiple possible degrees of use for a single instruction. The performance of each algorithm is first presented for a large predictor of fixed organization (8K-entry, eight-way set-associative) and complete tags (i.e., the entire instruction address is split between the set index and a per-entry tag). Smaller tags introduce the possibility of aliasing, the topic of Section 3.4.4, but the impact varies among the prediction algorithms, leading to different tagging requirements. With a suitable tag size chosen for each algorithm, the predictors can be compared in terms of capacity. Section 3.4.5 presents this comparison, demonstrating the conditions which favor particular predictor algorithms. The mechanisms of training and misprediction detection are revisited in Section 3.4.6 and Section 3.4.7, respectively. The issue of predictor bandwidth is addressed in Section 3.4.8.

3.4.1 Simple predictor: last observed degree of use

The data in Figure 2.10 demonstrate temporal locality in the per-instruction degree of use: 95% of all dynamic values have the same degree of use as the last value originating from the same static instruction. Therefore, a predictor maintaining only the last degree of use generated by each static instruction should be capable of 95% prediction accuracy. Coverage in such a predictor would be determined solely by the existence of an entry matching the address of the instruction.

The performance of this simple algorithm is shown in Figure 3.7. Similar presentations of predictor performance will appear throughout this chapter. From bottom to top, the stacked bars

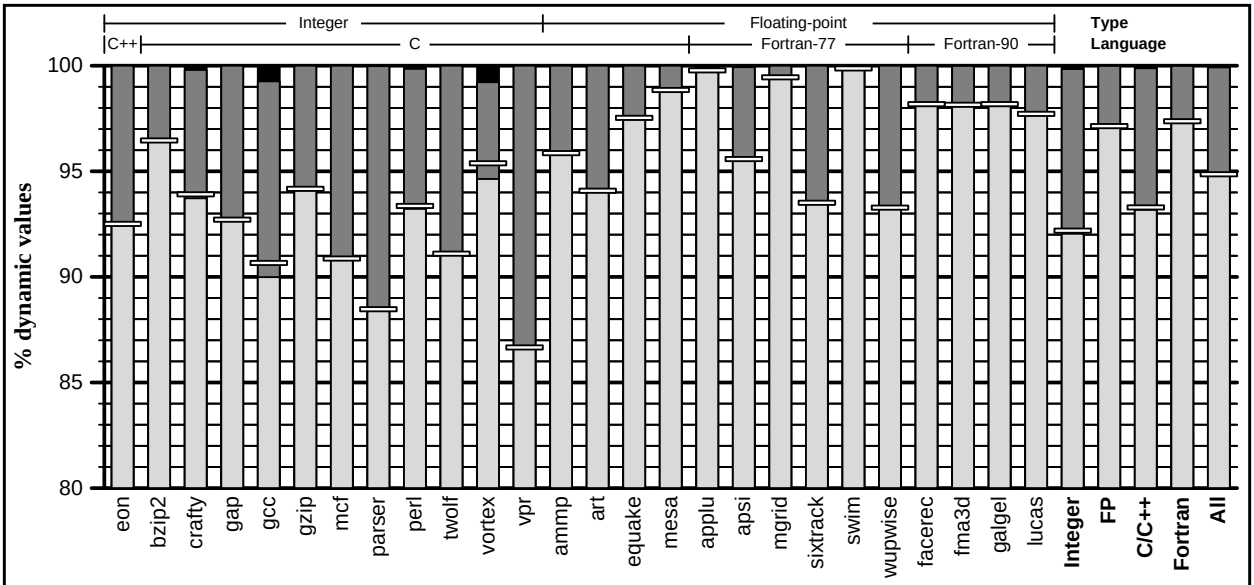


Figure 3.7. Performance of predicting last-observed degree of use

in the graph represent correct predictions (light gray), mispredictions (dark gray), and non-predictions (black) as a percentage of all dynamic values (in this particular figure, only a small number of non-predictions are visible, mostly in `gcc` and `vortex`). Coverage includes correct predictions and mispredictions and therefore may be read directly at the top of the dark gray bar. Accuracy is the height of the light gray bar (correct predictions) divided by the coverage. To enable easy visual comparison of accuracies across benchmarks, the accuracy has been superimposed on each bar as a small black-bordered, white hash mark. While the accuracy is read on the same vertical scale, it should be noted that accuracy is a percentage of predictions, not a percentage of all dynamic values. All predictor evaluations in this chapter were performed using timing simulation of the first four billion instructions of each benchmark. Details on the methodology may be found in Section A.4 of the appendix.

Returning to the results of Figure 3.7, at this predictor size, non-predictions are only detectable in a few benchmarks, and represent less than 1% of values in those cases. With coverage near 100%, accuracy is approximately the percentage of correct predictions, which averages 94.9%, matching expectations. The floating-point benchmarks exhibit an average accuracy better than 95%. Even the worst among them (i.e., `sixtrack` and `wupwise`) exceed the average prediction accuracy on the integer benchmarks. This behavior may be attributed directly to the smaller variability in degree of use behavior in the floating-point benchmarks noted in Chapter 2

and is due to their less complicated control-flow, both static and dynamic. Near perfect coverage is also achieved for all of the floating-point benchmarks, indicating small instruction working sets. This is true even for those benchmarks with static code footprints comparable to `gcc`, the largest of the integer benchmarks. Because this simplest instance of a degree of use predictor yields good results on the floating-point benchmarks, these benchmarks are omitted when developing the more sophisticated prediction algorithms of the next sections. The floating-point benchmarks will be revisited during the comparative evaluation in Section 3.4.5.

The average prediction accuracy on the more fickle integer benchmarks is only 92.2%. In particular, `parser` and `vpr` have accuracies less than 90%. Since aliasing is not a factor and a prediction is only made when a prior observation is stored for a static instruction, these benchmarks must be executing static instructions that exhibit poor temporal locality in their degree of use patterns. To improve the prediction accuracy, either predictions must be avoided on such instructions or additional information used to make the predictions.

3.4.2 Adding confidence

Always predicting the last observed degree of use suffers from the problem of being sensitive to temporary deviations from a dominant behavior. Consider an instruction with two different degrees of use, one of which occurs very infrequently. Obviously, when the rare degree of use occurs, a misprediction will result. However, upon the next execution, the predictor provides the infrequent degree of use, which most likely causes a second misprediction. Higher accuracy could be attained by retaining the more common degree of use rather than the most recently generated one. The potential for improvement can be seen by comparing Figure 2.10 with Figure 2.8. The data in the latter figure show that the most common degree of use could reduce the number of mispredictions by 24% (96.5% vs. 95.4% average accuracy).

The problem of retaining a more prevalent outcome—even when it is not the most recent one—exists for branch predictors. Just as for branch predictors, it can be solved by adding hysteresis in the form of saturating counters [77]. A saturating counter is associated with each predictor entry. The value of the counter is increased when the corresponding entry yields a correct prediction; otherwise, it is decreased. In this manner, the predictor can retain a dominant degree of use for an instruction even after the occasional misprediction. If the count reaches zero, however, the

stored degree of use will be replaced with a newly observed value. The value of the counter, then, indicates the confidence in the stored prediction.

Unlike the simpler last-observed-degree algorithm, this algorithm can choose not to supply a prediction based on a confidence threshold. The choice of this threshold in relation to the counter's maximum value (i.e., its range) provides another means to make the ubiquitous coverage-accuracy trade-off. In this manner, the predictor can exceed even the accuracy of one that *always* returns the single most likely degree of use by avoiding predictions for certain instructions. Thus, versus the last-observed-degree algorithm, this algorithm should convert some mispredictions into correct predictions (by virtue of returning more likely outcomes) while converting others into non-predictions. Both effects increase accuracy, although the second lowers coverage. The parameters of the confidence scheme will determine the magnitudes of these effects.

The parameters that define a confidence scheme are: (1) the range of the counter, (2) the adjustments made to a counter on correct and incorrect predictions, (3) the initial confidence of a new entry, and (4) the threshold below which no prediction is made. A counter value of zero always results in a replacement if it causes (or would have caused) a misprediction. To limit the number of designs considered, all schemes presented here use the full range of a two-bit counter and increase or decrease the counter value by one. The remaining free parameters are the initial confidence value and the non-prediction threshold.

Figure 3.8 shows the performance of the predictor versus the non-prediction threshold with the initial confidence fixed at 1. The S bar shows the performance of the simple predictor without confidence counters (i.e., the results from Figure 3.7) for comparison. The numbered bars represent a non-prediction confidence threshold of the indicated value. The presentation of each bar is the same as that of Figure 3.7 (i.e., correct predictions in white, mispredictions in gray, non-predictions in black, and accuracy as a white hash mark).

At all threshold values, accuracy is improved substantially over the simple predictor. The effect of the added hysteresis is clearly evident at low threshold values by the increased number of correct predictions versus the simple predictor. As the threshold is increased, the predictor becomes more selective, making fewer predictions to increase accuracy.

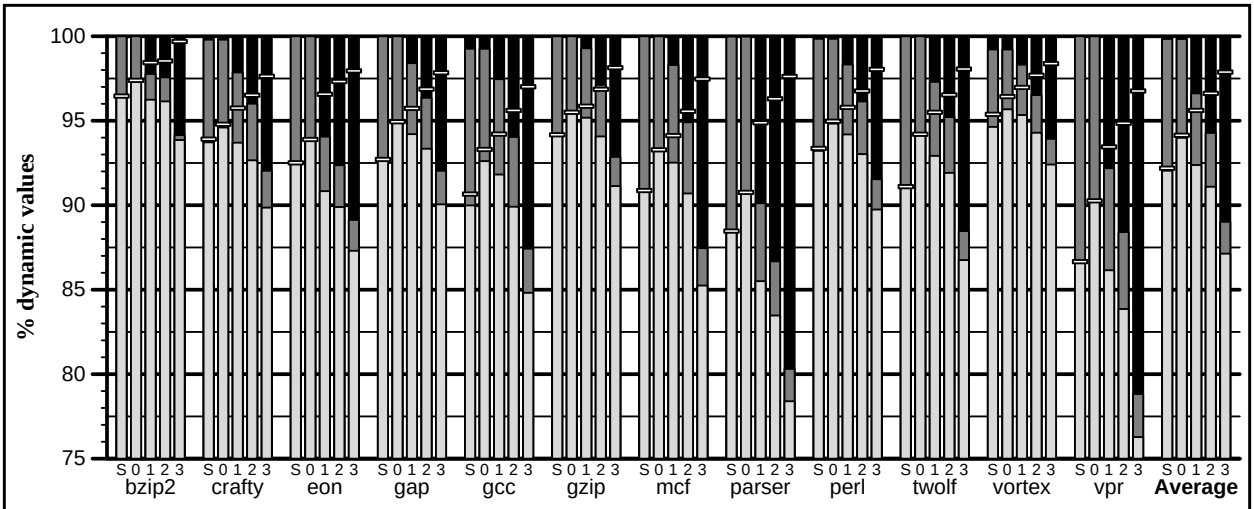


Figure 3.8. Performance vs. non-prediction threshold for predictor with confidence counters

When the threshold equals the maximum confidence, the benefit of the confidence scheme is due entirely to its enhanced selectivity—at this point the hysteresis cannot result in a prediction that would not have been made by the simple predictor, reducing the number of correct predictions substantially. To explain this phenomenon, consider a particular static instruction that yields degrees of use 1-2-1-1 during consecutive executions and simple and confidence-enhanced predictors both pre-trained to a degree of use of 1 (at maximum confidence) for the instruction in question. The simple predictor will predict 1-1-2-1, for a total of two mispredictions and two correct predictions. The enhanced predictor with a threshold equal to the maximum confidence will return 1-1-N-1, while lowering the threshold by one will result in the sequence 1-1-1-1. In both cases, only one misprediction occurs, but only with the lower threshold is the number of correct predictions increased over the simple scheme.

The effect of the initial confidence, portrayed in Figure 3.9, is weaker than that of the non-prediction threshold and operates in the opposite direction (i.e., higher values of this parameter decrease accuracy and increase coverage). In this figure, the numbered bars indicate the initial confidence values. As before, the S bar shows the performance prediction of the last-observed degree of use. A non-prediction threshold of 2 was selected for this experiment to avoid sacrificing the hysteresis benefit. In the steady state, most instructions will be present in the predictor. Assuming adequate capacity, replacements only occur when the instruction working set changes significantly. The initial confidence assigned to these new observations determines how quickly the predictor will react to the new conditions. The lower the initial confidence, the more closely

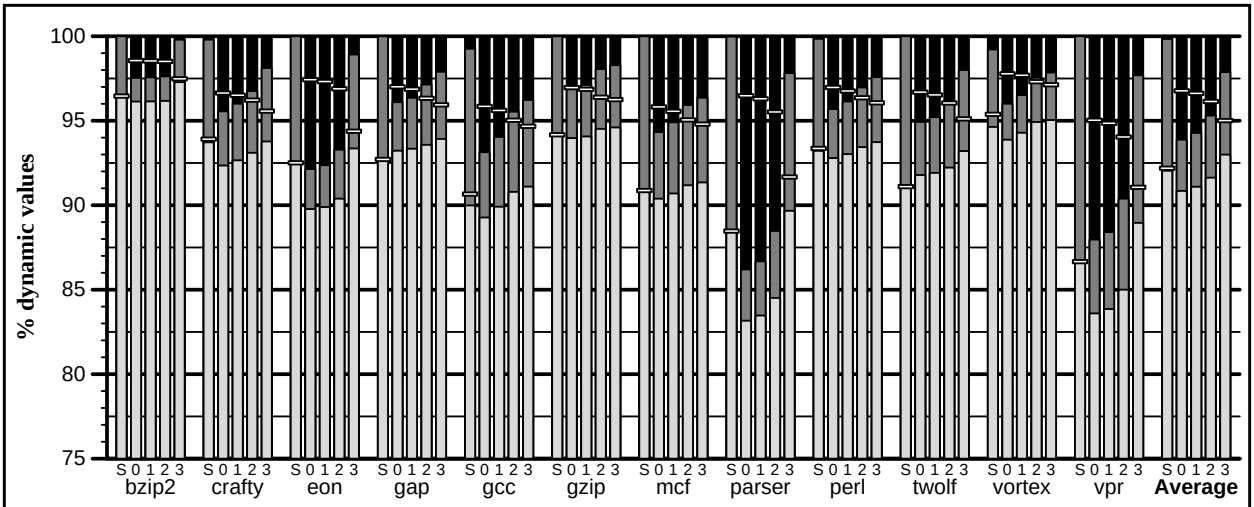


Figure 3.9. Performance vs. initial confidence for predictor with confidence counters

the predictor state will track rapidly changing conditions, such as might be expected during a phase change.

For the remainder of this chapter, an initial confidence of 0 and a non-prediction threshold of 2 are assumed. These values correspond to the 0 bar in Figure 3.9. The confidence-enhanced predictor makes 6.1% fewer predictions than the last-observed-degree predictor. As many of these non-predictions were mispredicted by the simpler predictor, the average accuracy is increased from 92.2% to 96.8%.

3.4.3 Using control-flow information

Consider the nature of the mispredictions produced by the previous two prediction algorithms. Neither of the previous algorithms has the option of issuing a different prediction than the single stored degree of use. Thus, the mispredictions must occur on instances of static instructions that can generate multiple degrees of use. The confidence scheme can avoid making predictions for these instructions where their dynamic behavior is irregular, but it can never generate correct predictions for consecutive instances of an instruction that have different degrees of use, limiting the best-case rate of correct predictions.

A prediction algorithm capable of exceeding this limit requires a means to distinguish among multiple degrees of use for a single static instruction. Making this distinction necessitates additional information for the generation of a prediction. Such information must be available to the predictor before the prediction is needed and must also differentiate among various dynamic cir-

cumstances that lead to different behaviors from a single static instruction. An obvious candidate that meets both of these requirements is the history information maintained by the instruction sequencer for branch prediction.

The branch history provides information about the dynamic flow of execution leading up to the instantiation of a particular instruction. This information provides a context within the framework of the program that can imply that a particular dynamic instruction will behave in a certain manner. Applied to branch prediction, this control-flow context is used to decide the direction of an instance of a static branch. In a similar fashion, this information can be used to decide the number of uses likely for a certain instance of a static instruction.

Branch history information works in this role precisely because it is correlated with upcoming branch outcomes. The degree of use of a value is completely determined by the instructions that are encountered *after* the value is generated. These instructions are in turn determined by the future control flow. Therefore, branch history information, which can help predict the expected path of execution, can also discriminate among different potential degrees of use for a value.

Rather than using the branch history, it is also possible to directly use the branch future due to pipelining. Instructions in the middle of a pipeline have available to them the predicted outcomes of control instructions occurring later in the dynamic instruction stream (i.e., earlier in the pipeline). So long as the branch predictions are used in a later pipeline stage than where they were generated, they represent the future. Given a high branch prediction accuracy, these predictions are equivalent to path look-ahead, providing nearly perfect knowledge about the uses occurring in the immediate future. The future control-flow information need not be restricted to conditional branch outcomes. If an indirect branch (e.g., a procedure return) occurs on the future path, its predicted target address may also be used.

A reasonable question to ask is why using future control flow would provide any better performance than a scheme that directly takes in the branch history. After all, the branch predictor itself turns branch history information into the branch predictions that such a scheme would subsequently use. Recall that values with low degrees of use (i.e., most values) are completely used within a short distance of when they are generated (see Figure 2.4). Thus, most of the time, only a few upcoming branch directions are required to uniquely determine the number of uses a value will see. Prior studies have shown that a given branch's outcome may be correlated to one very

far in the past [30]. Thus, a long branch history may be needed to have the same information about the immediate forward path. Using more input bits (in the form of a long branch history) implies more variation and a larger predictor in order to correlate this input information with a particular degree of use. Also, the branch predictor is specialized: its very structure encodes knowledge about the ways branches behave. Since the branch predictor already an accurate distillation of long histories into expected outcomes, it is not necessary to have a large degree of use predictor perform the same task less efficiently.

Exploiting control-flow information involves modifying the predictor to allow multiple entries per static instruction. The generation of a prediction for an instruction (the *target*) commences by using a portion of the instruction address to access a set within the predictor as in the previously presented algorithms. However, by extending the tag to include the control-flow information, multiple predictor entries—possibly with different degrees of use—can coexist for a single static instruction. The portion of the tag containing the encoded control-flow information is referred to as the *control-flow signature*. In order to select a particular entry within the set, the entire tag must be matched, corresponding to a match of both the static instruction and its associated dynamic control-flow context. Using the control-flow information late in the prediction process facilitates signatures based on future control flow. Note that this algorithm requires a set-associative predictor in order to be able to store multiple predictions per static instruction.

Figure 3.10 illustrates the three different types of signatures that are evaluated here. The history method uses the most recent bits of the global branch history at the time of the fetch of the target instruction as the signature. The future signature consists of the predicted directions of conditional branches between the front end of the machine and the target instruction. indirect is

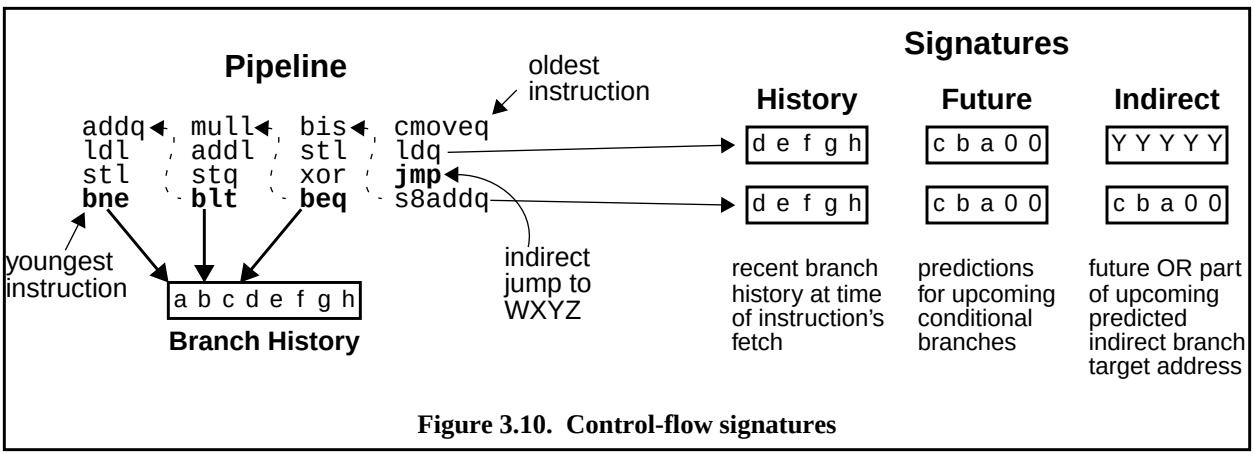


Figure 3.10. Control-flow signatures

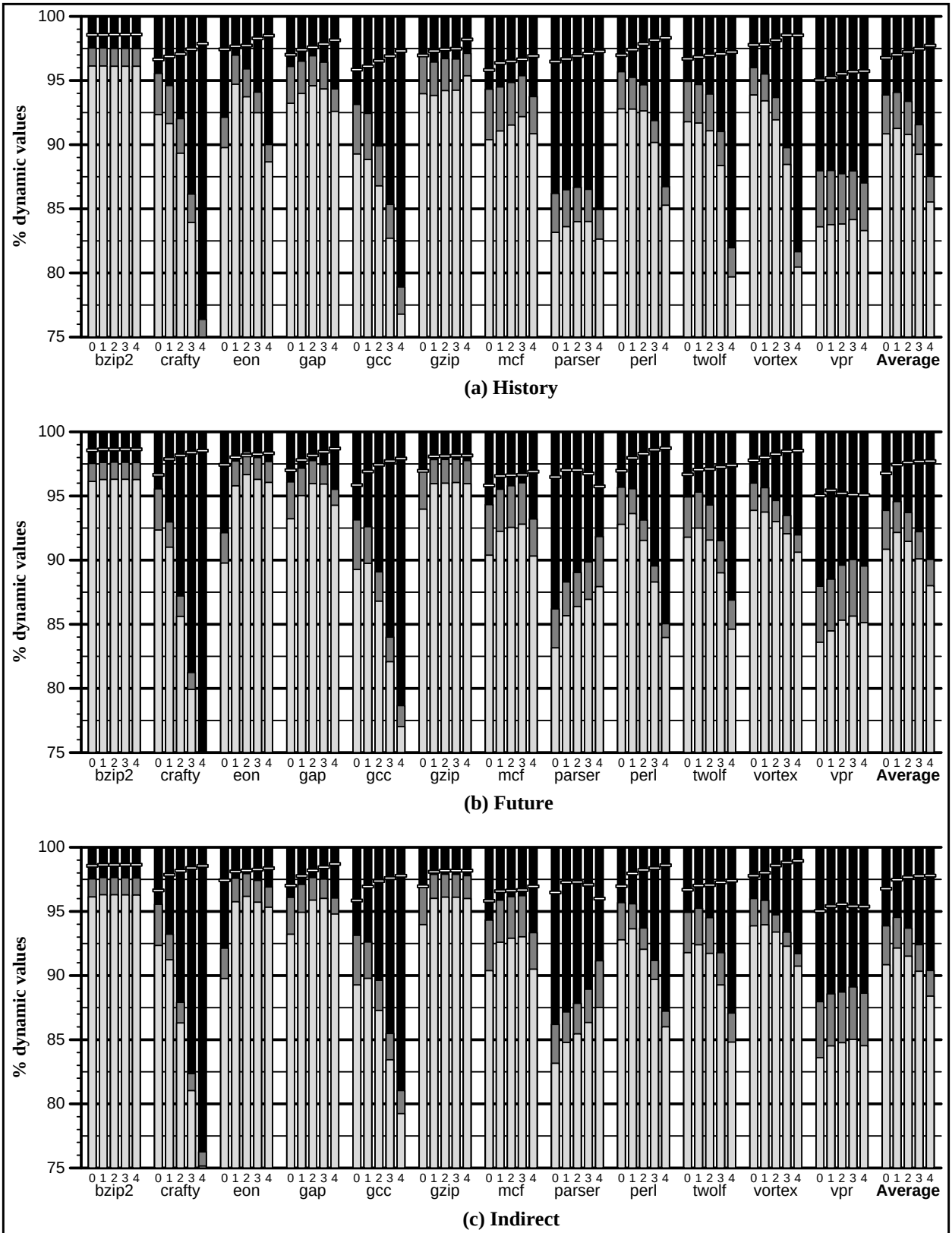
identical to future except when the first control instruction after the target is an indirect jump (including a procedure return); in this case the signature is the low order bits of the instruction cache index of the jump's target.

Figure 3.11 shows the performance of the predictor as a function of signature length for each signature type. The left-most configuration for each benchmark (the 0 bars in the figure) shows the performance of a predictor without a control-flow signature for comparison. The higher numbered bars reflect a signature with the indicated number of bits. All of the predictors use the confidence mechanism described in Section 3.4.2.

For all of the algorithms, longer signatures improve the average accuracy. In the case of the history signature, the accuracy increases steadily over the range of signature lengths considered. In the case of the future and indirect signatures, most of the accuracy benefit is obtained with only two signature bits. Adding signature bits increases the possible number of entries for a single static instruction exponentially. This increases capacity pressure on the predictor, reducing the coverage significantly for most benchmarks at long signature lengths.

Interestingly, for up to two signature bits (one in the case of the history signature), the average correct prediction rate is improved over the baseline predictor without any control-flow. For certain benchmarks (e.g., gap and parser), this improvement continues to longer signature lengths. These additional correct predictions come from instances of instructions with different degrees of use. In the baseline predictor, which can maintain only one degree of use, the variable behavior lowers the prediction confidence for these instruction below the non-prediction threshold.

Comparing the history signature versus the forward control-flow based signatures reveals the advantage of these latter schemes. For shorter signatures, the future branches are more likely than the past branches to select the correct degree of use resulting in higher accuracy. As the signature length increases, the accuracy of the history scheme improves becoming comparable at a signature length of four bits. At this point, however, the coverage is substantially lower. Again, this may be attributed to a better correlation between the future branch directions and the degree of use of a given instruction. In cases where the long history signature is still unable to differentiate instances of an instruction having different degrees of use, the confidence mechanism will come



(a) History

(b) Future

(c) Indirect

Figure 3.11. Degree of use predictor performance as a function of signature length

into play, reducing the number of predictions made. Of the two forward control-flow based signatures, the indirect signature performs just slightly better on average.

The biggest disadvantage of the control-flow based predictors described is the reduction in coverage due to signature variability. Each different signature encountered for an instruction requires its own predictor entry, resulting in significant capacity pressure. Coverage is impacted the most for those benchmarks with highly variable control flow (e.g., `crafty` and `gcc`). Since most instructions only exhibit a single degree of use, these multiple entries are wasteful. A simple optimization to take this common-case behavior into account reduces this pressure substantially, allowing a longer signature to be used without unduly affecting the coverage.

The modification involves selectively ignoring the signature altogether and matching only on the address portion of a tag. To each entry, a single bit, the *easy bit*, is added to the signature. When set, the signature portion of the stored entry is ignored, allowing a match with a specific static instruction regardless of the control flow. On a write miss (i.e., an insertion), the easy bit is set only if there is no other entry in the set with the same address portion of the tag. If a write hit occurs to an entry with the easy bit set and the stored degree of use differs from the one being trained, the stored entry is immediately replaced and the easy bit is cleared. Normally in this event, the confidence would be decreased instead and the replacement would only occur when it reached zero. The modified policy ensures that a control-flow signature is associated with a particular degree of use outcome as soon as multiple outcomes are known to occur.

The performance of the enhanced predictor versus signature length is shown in Figure 3.12. The I bar (I not 1) within each group is the baseline predictor using a two-bit indirect signature, which was among the best performing predictors from Figure 3.11; an indirect signature is also used along with the easy bit enhancement in the other configurations. The digits below the other bars indicate the number of bits used in the indirect control flow signature. Thus, the 2 bar is the same as the I bar in every respect except for use of the easy-bit modification.

As expected, adding the easy bit increases the coverage, providing a higher number of correct predictions. This increase is achieved without sacrificing accuracy because only those instructions that have a fixed degree of use avoid the requirement for a control-flow signature match. With this enhancement, a control-flow signature of three or four bits is preferred over the two-bit signature selected previously. For the remainder of the chapter, an indirect signature of three bits

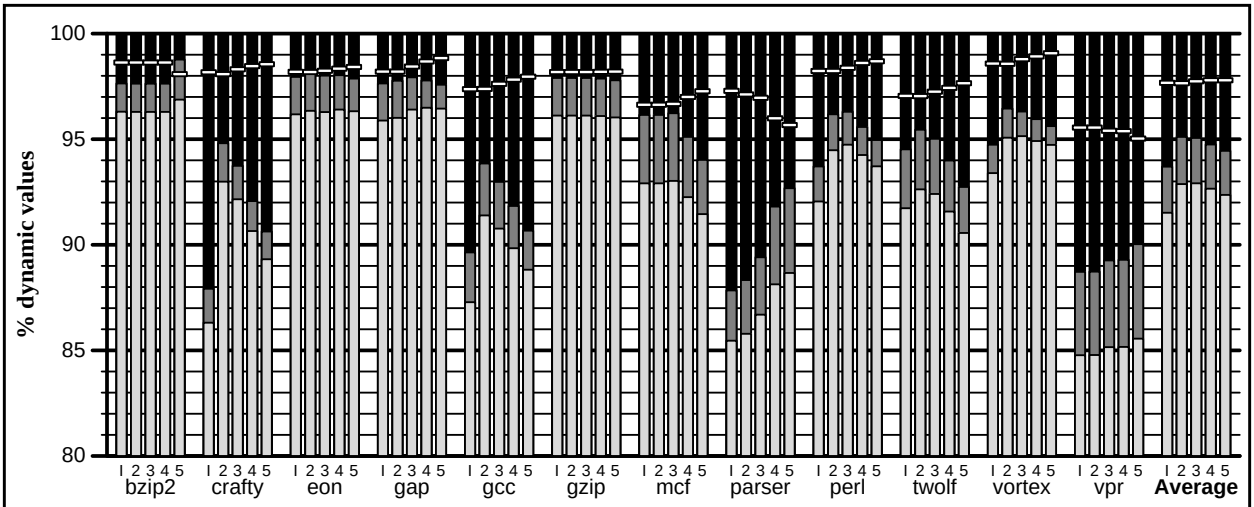


Figure 3.12. Easy-bit enhancement to control-flow signature

plus an easy bit will be assumed as the configuration of the control-flow enhanced predictor (represented by the 3 bars in Figure 3.12).

Comparing Figure 3.12 with Figure 3.7 shows that versus the simple last-observed-degree predictor, the control-flow enhanced predictor reduces the number of mispredictions by 72% while simultaneously increasing the number of correct predictions. This benefit comes at the cost of adding six bits (two confidence bits, three signature bits, and the easy bit) to each three-bit degree of use entry in the predictor. Adding the additional storage to the simple predictor as presented in Section 3.4.1 would not have helped since it already had nearly perfect coverage and the accuracy would not have changed. In Section 3.4.5, the effect of smaller capacity limits are considered. First, however, the size of the address portion of the tag must be determined.

3.4.4 Aliasing in degree of use predictors

Aliasing occurs when a predictor is unable to distinguish among multiple, distinct static instructions. Since each static instruction is uniquely identified by its address, aliasing can result when predictions are associated with anything less than the full instruction address.† Destructive aliasing results when the behavior of two or more indistinguishable instructions differs. In this situation, predictor accuracy can be reduced if the predictor supplies the wrong prediction for one instruction based on the stored information about another instruction. Destructive aliasing is a

† In a multi-programmed environment, of course, static instructions belonging to different programs in different address spaces may share the same virtual address and, at times, physical address.

well-studied phenomenon in the area of branch prediction because it is the primary cause of reduced predictor accuracy. There is a greater potential for destructive aliasing in degree of use prediction because of the multi-valued nature of the predictions.

While the solution to destructive aliasing is simple—using enough instruction address bits to differentiate stored predictions, as has been assumed to this point—it has an associated hardware cost due to the large tags required in the predictor. The tag sizes may be reduced provided the incidence of destructive aliasing is low. Note that the frequency of aliasing depends on the *total* number of address bits used to select a prediction, including both the tag and the set index. Thus, between two predictors with the same capacity and tag size, the more associative predictor will suffer more from aliasing (since it has fewer set bits).

Once aliasing becomes significant, the coverage increases (as predictions are generated for instructions that are not actually present) while the accuracy drops. The coverage increase due to aliasing is not really beneficial, however, since it will contain a much higher percentage of mispredictions than the overall rate (aliasing is more likely to be destructive). The accuracy decrease can be reduced or delayed given a predictor that can use additional information besides the address to differentiate aliased instructions.

Table 3.1 provides data on the occurrence of aliasing versus the number of address bits for the prediction algorithms of the preceding sections. The table presents the average alias rate (over the integer benchmarks) in terms of the number of predictions generated based on stored data for a different instruction. The incidence of destructive aliasing is presented as a percentage of the

Table 3.1: Aliasing Rates

Tag bits	Total bits	Alias rate (% predictions)			Destructive alias rate (% aliased preds)		
		simple	confidence	control-flow	simple	confidence	control-flow
3	11	89.54	89.49	18.19	14.16	7.41	4.50
4	12	35.19	35.33	11.35	14.16	7.98	3.80
5	13	13.20	12.75	4.83	15.72	8.94	4.04
6	14	5.36	5.29	2.18	13.33	7.84	4.00
7	15	2.76	2.77	1.15	10.61	5.43	3.42
8	16	0.87	0.84	0.37	8.88	5.58	3.62
9	17	0.33	0.31	0.19	4.77	2.73	1.64

aliased predictions that are incorrect. A smaller predictor (256-entry, eight-way set-associative) has been used to increase the possibility of aliasing. Thus, the total number of address bits available to distinguish any two instructions is the indicated number of tag bits plus the eight bits used to select a particular set. A minimum of three tag bits are required in this configuration because there are eight entries per set.

The alias rate for the simple and confidence predictors is nearly identical and drops by roughly a factor of 2.5 per extra tag bit. The advantage of the confidence predictor with respect to aliasing is evident in its lower destructive aliasing rate, however, which is just over half of the simple predictor's. Employing a control-flow signature significantly reduces the occurrence aliasing due to the availability of the signature to help distinguish instructions. The frequency of destructive aliasing is also reduced, providing additional benefit.

The overall effect of aliasing on the prediction accuracy is shown in Figure 3.13 (for the original 1K-entry, eight-way set-associative predictors). The accuracy of the control-flow predictor never drops below even the unaliased accuracy of the confidence-only configuration. Similarly, the confidence predictor always provides better accuracy than the simple predictor. Therefore, part of the storage cost of the more complicated algorithms can be offset by the need for fewer tag

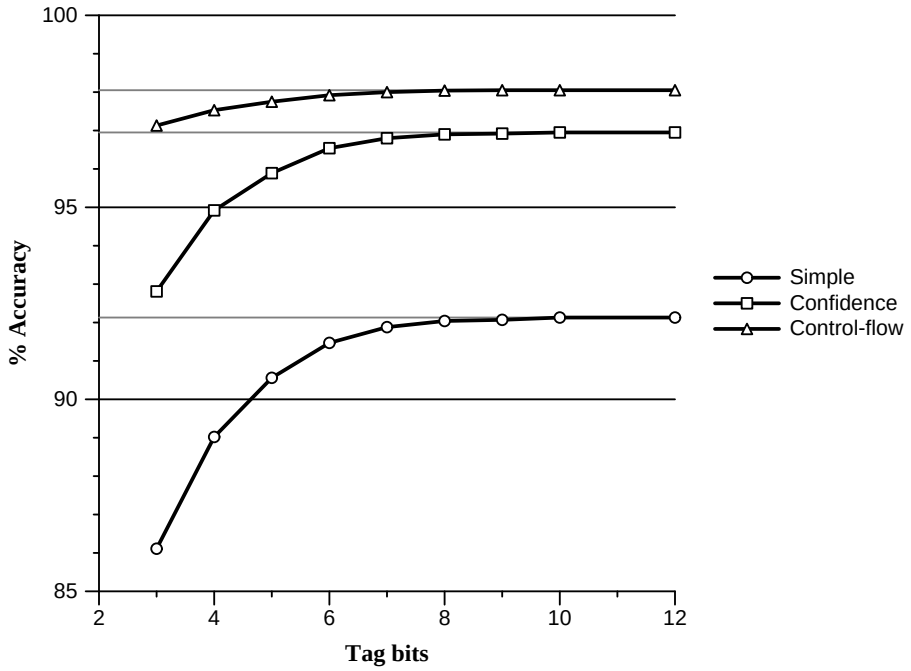


Figure 3.13. Effect of tag length on predictor accuracy

bits. For example, the simple predictor with a nine-bit tag has the same number of bits per entry (12) as the confidence-based predictor with a seven-bit tag and the control-flow predictor with a three-bit tag. In spite of the additional aliasing suffered by the more sophisticated algorithms, the relative ordering of their performance is unchanged. Since the cost of the added storage is less likely to be a factor than the question of how to effectively increase the prediction accuracy with any amount of storage, the tag length is instead chosen for each algorithm based on when aliasing begins to noticeably affect accuracy. Based on the curves in Figure 3.13, this occurs at six tag bits for the control-flow enhanced predictor and seven tag bits for the other two predictors. Tag lengths that (with the set index) yield the same total number of address bits (16 or 17) are assumed for the remainder of the chapter.

3.4.5 Comparative evaluation

To this point, the performance of each prediction algorithm has been studied using an 8K-entry predictor, which is large enough that capacity limitations did not significantly impact the results. In this section, the performance of the prediction algorithms will be compared at different capacities and the conditions that favor the different algorithms revealed.

Figure 3.14 shows how coverage depends on predictor size and associativity. Not surprisingly, higher capacity yields higher coverage. At the largest predictor size shown, the simple predictor delivers nearly 100% coverage (also shown in Figure 3.7). Increasing the coverage for the other two algorithms even slightly requires significant extra capacity.

The importance of associativity is clearly evident in the figure. For the simple and confidence algorithms, which do not employ control-flow information, there is a one-to-one correspondence between instructions and predictor entries. Thus, the only purpose of increased associativity in these predictors is to reduce conflicts; as a result, the benefit of higher associativity declines with predictor capacity. Adding control-flow signatures causes some instructions to occupy many entries within the same set, greatly increasing the importance of associativity. Evidence of this detail is visible in the distance between the curves corresponding to the control-flow enhanced predictor. Longer signatures exacerbate this dependence by increasing the number of entries that can be associated with a single static instruction. Another effect of allowing multiple entries per

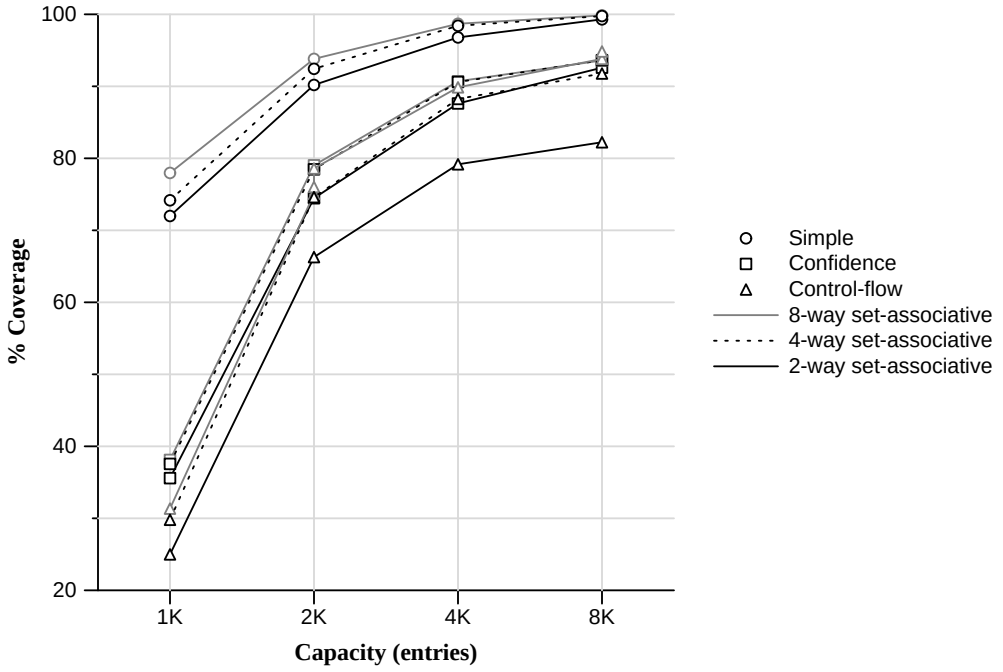


Figure 3.14. Predictor coverage vs. organization

instruction is increased capacity pressure, which is evident in the steeper slope of the curves for the control-flow predictor.

Figure 3.15 shows the interaction of accuracy and coverage for the predictor configurations of Figure 3.14. The shape of each mark indicates the prediction algorithm, while the mark's color indicates the associativity. Capacity is not indicated explicitly, although groups of marks corresponding to different associativities at a given capacity (1K, 2K, 4K, or 8K entries) are visually-separable except for the simple predictor.

To first order, accuracy is independent of a predictor's size since the prediction made depends only on the contents and not on the availability of an entry. In other words, accuracy is primarily dependent on the prediction algorithm while the coverage depends on the predictor's capacity. This property motivated the definition of coverage to include all predictions made instead of just including correct predictions.

The control-flow enhanced predictor exhibits the largest variation in accuracy with capacity decreasing from 98.5% for the 1K-entry, two-way predictor to 97.7% for the 8K-entry, eight-way predictor. To explain this variation, first note that similar variation is seen at constant capacity as the associativity changes. Lowering the associativity reduces the ability of this predictor to main-

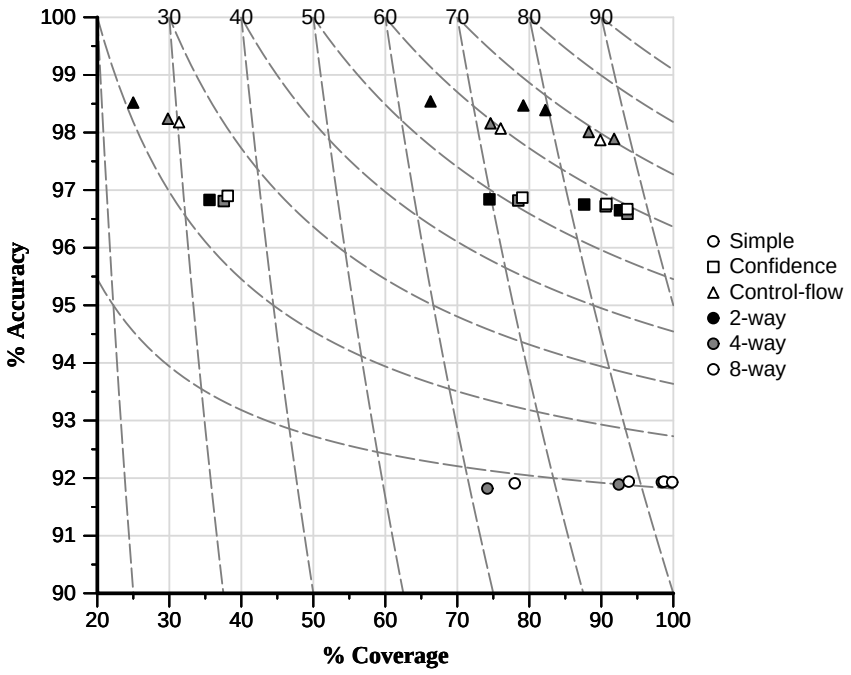


Figure 3.15. Prediction accuracy vs. coverage

tain all possible signature variants for those static instructions with multiple degrees of use. This lowers the coverage but *increases* the accuracy since these are the instructions that are hard to predict. Reducing the capacity has the effect of increasing the competition for entries within a set, which causes the same accuracy increase (and coverage decrease) as directly reducing associativity.

Superimposed on Figure 3.15 are dotted-line contours representing constant benefit (using the simple model described by Equation 9 of Section 3.1). The near-vertical contours correspond to a cost factor of one while the curved ones represent a cost factor of ten. At low cost factors, benefit is more strongly dependent on coverage. Thus, even the simple predictor delivers benefit comparable to that of the larger, more complex predictors. At low capacities, the simple predictor is significantly better. As the cost of mispredictions increases, accuracy plays a more important role and the more complex predictors are favored. At a cost factor of ten, 2K-entry control-flow enhanced predictors deliver benefit on par with confidence-only predictors four times larger. Though the coverage is around 20% lower, the 1.5% increase in accuracy is more important. When mispredictions are this costly, the simple predictor cannot compete with any of the other predictors regardless of its capacity.

Heretofore, capacity has been expressed in terms of entries, which unfairly handicaps the simpler predictors with their smaller entries. In order to perform a true comparison based on capacity, all bits within the predictor storage must be accounted for. The prediction algorithm specifies the number of bits in each entry, and the choice of tag length was already addressed in Section 3.4.4. However, the use of set-associative storage requires a replacement policy, which may require additional state. In the preceding sections of this chapter, perfect LRU replacement was assumed. Implementation of this strategy for an eight-way set-associative predictor requires a minimum of ceil($\log_2 8!$) = 16 state bits per set to maintain ordering among the entries; a more reasonable encoding (with respect to the update logic) requires $8 \times (8 - 1) \div 2 = 28$ bits. Even for the control-flow enhanced predictor, this represents a 23% storage overhead just to implement the replacement policy.

A sub-optimal replacement policy manifests as lower predictor coverage (since more worthwhile entries will occasionally get evicted over ones less so). As the associativity increases (and LRU becomes more expensive), the importance of the replacement policy diminishes and rougher approximations of LRU suffice without a significant impact. Other possible replacement policies for an *n*-way set-associative predictor include tree-based pseudo-LRU ($n - 1$ bits/set), not-MRU ($\log_2 n$ bits/set), and random (free).

Figure 3.16 compares these policies on an 8K-entry, eight-way set-associative predictor. The control-flow prediction algorithm is used since its performance is most sensitive to associativity.

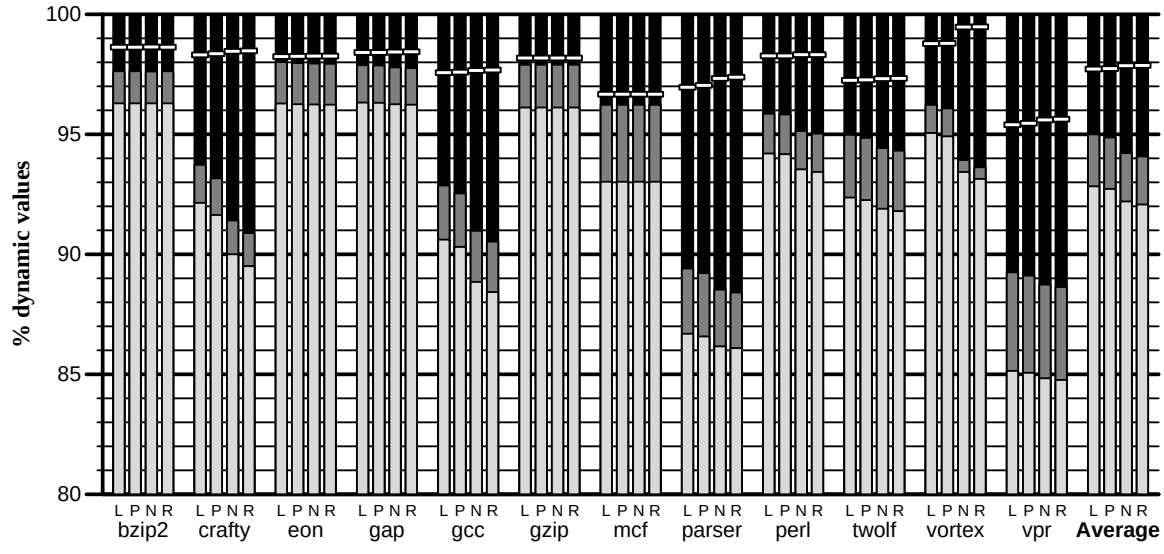
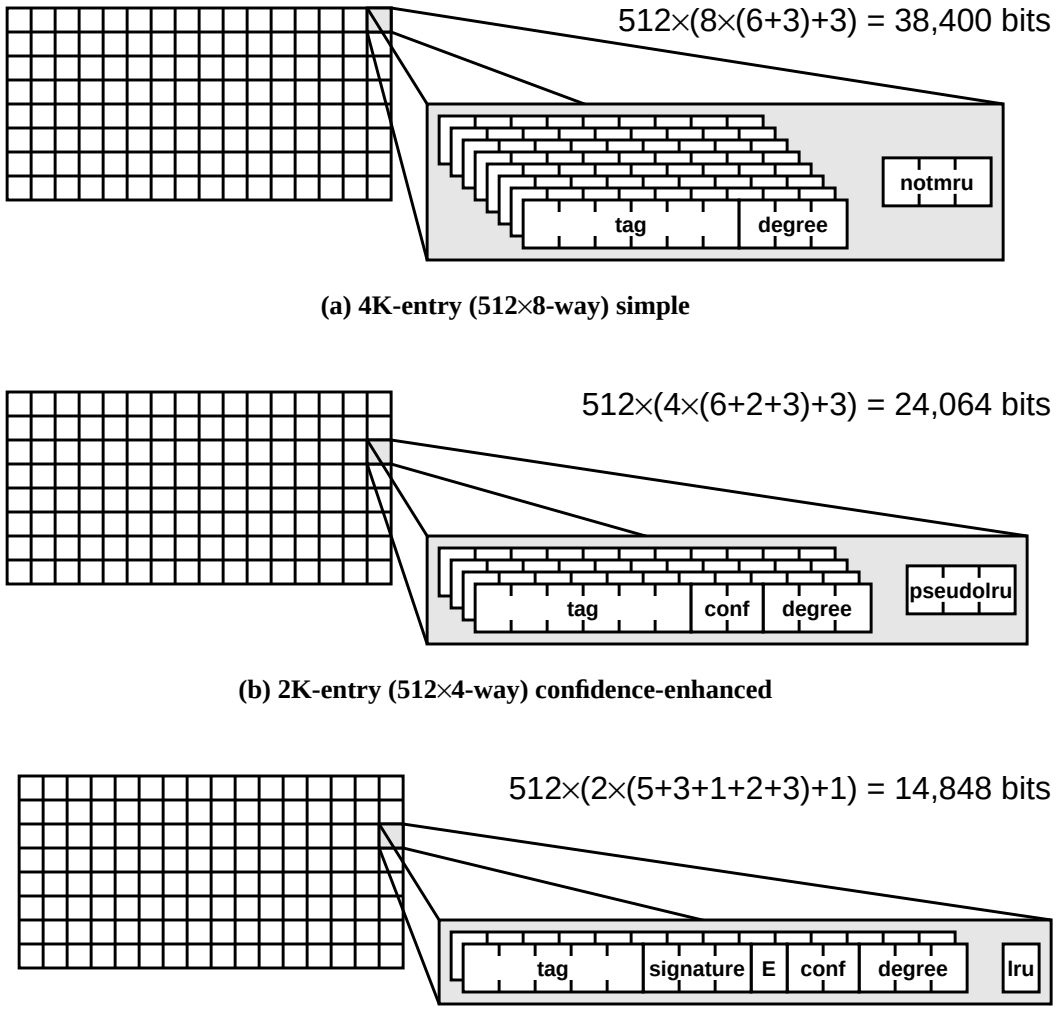


Figure 3.16. Comparison of replacement policies

From left to right, the bars represent LRU, Pseudo-LRU, not-MRU, and Random replacement. Pseudo-LRU performs similarly to true LRU, while not-MRU and random show noticeably lower coverage. Interestingly, due to the slight increase in accuracy accompanying the loss in coverage, the not-MRU scheme is preferred to pseudo-LRU for cost factors greater than four. Given that it also has less than half of the space overhead (three vs. seven bits/set), not-MRU replacement will be used for eight-way predictors. At four-way set-associativity, pseudo-LRU replacement offers the best average performance through cost factors exceeding 11 with only one bit/set more than not-MRU. Finally, true LRU is feasible at two-way associativity. These replacement policy choices are reflected in the data of Figure 3.14 and Figure 3.15.

With a replacement policy decided, the capacity of a given predictor organization can be computed. Figure 3.17 illustrates the contents of a representative predictor of each type, demonstrating the differing capacity requirements of each. All predictors must store observed degrees of use. Beyond that minimum requirement, algorithm-specific storage consists of a confidence counter, control-flow state, and the easy bit (labeled E in the figure). As discussed in Section 3.4.4, the number of tag bits depends on number of sets in the predictor such that a certain total number of address bits are used in generating a prediction. Finally, the choice of associativity dictates the storage overhead of the replacement algorithm.

Figure 3.18 plots the benefit of degree of use prediction versus the capacity of the predictor. Each of the predictor configurations from Figure 3.15 is represented. Again, cost factors of one (solid lines) and ten (dotted lines) are illustrated. The best design for a fixed amount of storage depends on the cost factor: low cost factors will favor simple designs with higher coverage while high cost factors will favor fewer predictions of higher accuracy. With a large enough hardware budget, the coverage of the complex predictors is raised to the point that they become competitive even at a low cost factor. These trade-offs are clearly illustrated in figure. For example, at a cost factor of one, the simple predictor is preferred until the hardware budget exceeds 4K-bytes. The confidence-based predictor then reigns until control-flow predictors of around 12K-bytes are possible. As the cost factor increases, the complex predictors are preferred even at low capacities. For a cost factor of 10, the control-flow predictor delivers the best performance down to less than 3.5K-bytes.



(a) 4K-entry (512×8-way) simple

(b) 2K-entry (512×4-way) confidence-enhanced

(c) 1K-entry (512×2-way) control-flow-based

Figure 3.17. Degree of use predictor contents

Several circumstances favor the selection of the control-flow enhanced prediction algorithm. First, hardware budget is not likely to be a constraint. The largest predictor represented in Figure 3.18 has a capacity less than 16K-bytes, which is small compared to many contemporary proposals for branch predictors. While the utility of branch predictors is most certainly larger, a degree of use predictor is only likely to be used when extra transistors do not add much marginal benefit in more traditional roles (e.g., cache capacity). Second, the latency of the predictor's storage is not critical, removing another constraint on its size. Predictions cannot possibly be used until the corresponding instructions are available. Since the actual storage access requires only an instruction address (the control flow information being used later), it may proceed in parallel with

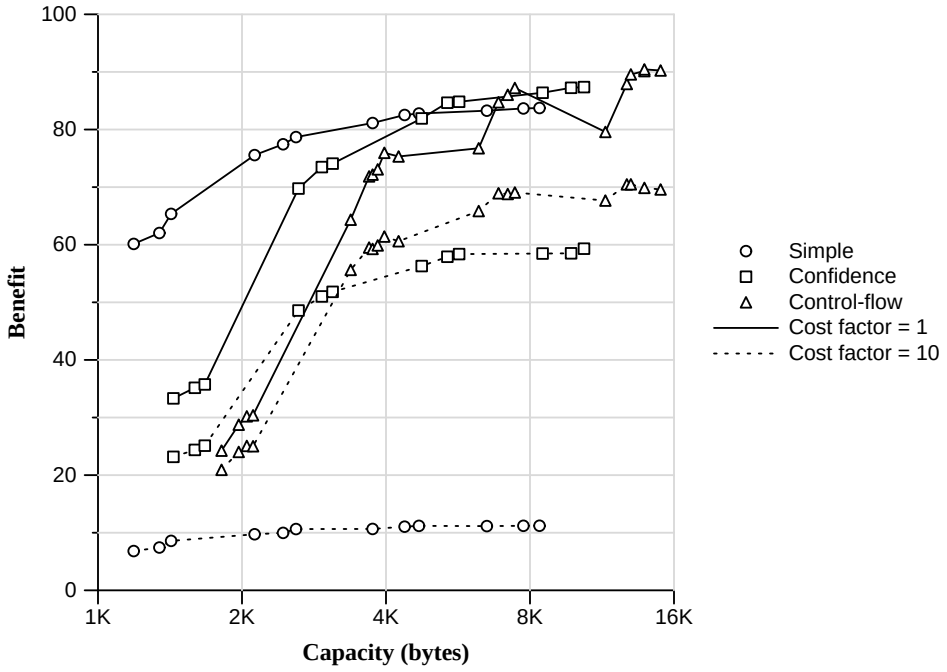


Figure 3.18. Benefit of different prediction algorithms vs. capacity

the fetch of the corresponding instruction. Finally, given adequate storage, the control-flow enhanced prediction algorithm is robust with respect to the cost factor—whether mispredictions are relatively cheap or expensive, it offers the most benefit.

Based on these considerations, the large control-flow enhanced predictor represented in Figure 3.18 (and also Figure 3.14 and Figure 3.15) will be used as the degree of use predictor in the remainder of the dissertation (unless otherwise noted). To reiterate the parameters of this particular predictor, it is an 8K-entry, eight-way set-associative control-flow enhanced predictor. Each entry consists of a five-bit tag, a three-bit indirect future control-flow signature, an easy bit, a two-bit confidence counter, and a three-bit degree of use. The non-prediction threshold is two and the initial confidence of a new entry is zero. The replacement policy is not-MRU, yielding a total capacity of 13.4K-bytes.

Figure 3.19 presents the performance of this predictor on all 26 of the SPEC benchmarks. Comparing with Figure 3.7 demonstrates the magnitude of the improvement achieved over the simple predictor. Average accuracy on all benchmarks has improved to nearly 99%, corresponding to a 77% drop in misprediction rate. In spite of an increase in non-predictions, the average correct prediction rate has actually increased slightly. Optimization of the predictor on the more

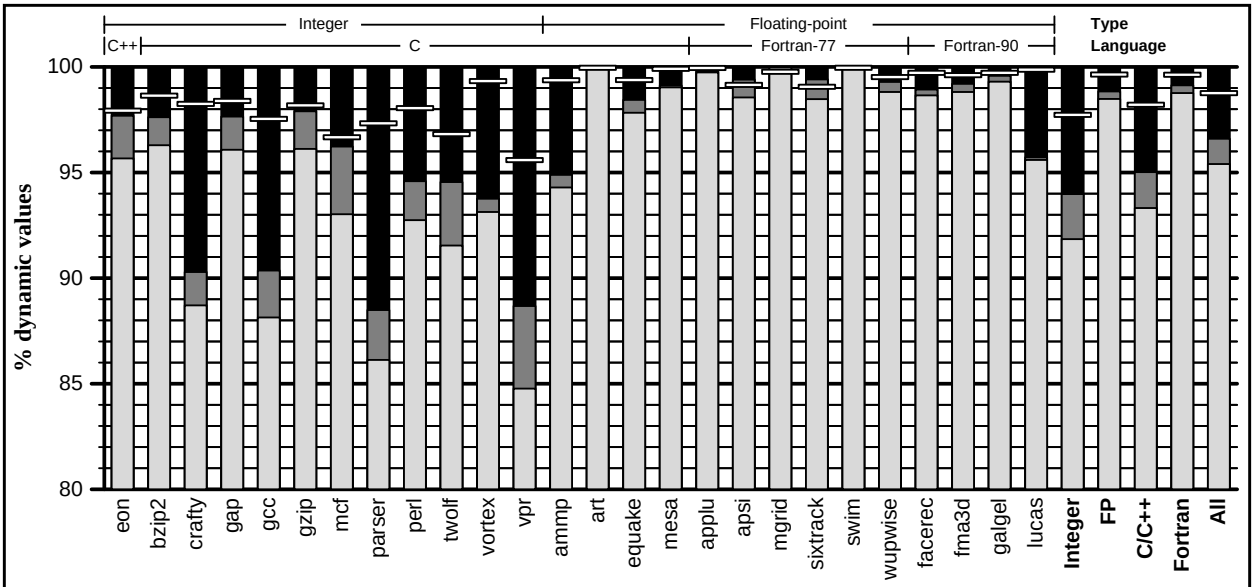


Figure 3.19. Tuned predictor performance on all benchmarks

difficult integer benchmarks does not appear to have negatively impacted its performance on the floating-point benchmarks, none of which shows an accuracy less than 99%. With respect to the integer benchmarks, all prediction accuracies have been brought above 95% with coverages greater than 87%.

Prediction accuracies presented so far have been aggregate accuracies over all degrees of use. Figure 3.20 breaks down predictor performance by predicted degree. Above each bar is the percentage of all predictions accounted for by that degree of use (the usual caveat about rounded percentages not summing to 100% applies). Degree of use one values are predicted with very high accuracy—over 99% on average and nearly that for even the integer benchmarks. Although degree of use one predictions are the most accurate, they contribute more total mispredictions than any other degree of use due to the large number of predictions involved (68% of all predictions). The prediction coverage of single-use values is also higher than for any other category. This behavior may be attributed to the fact that most single-use values—most values overall, in fact—tend to be dedicated temporaries used in direct communication between a pair of static instructions, which is trivial to predict with perfect accuracy.

Accuracy degrades with increasing degree of use in the C/C++ benchmarks. Values with higher degrees of use are more likely to exhibit variable behavior, rendering them more difficult to predict. Also, their longer lifetimes increases the possibility that branches beyond the look-ahead

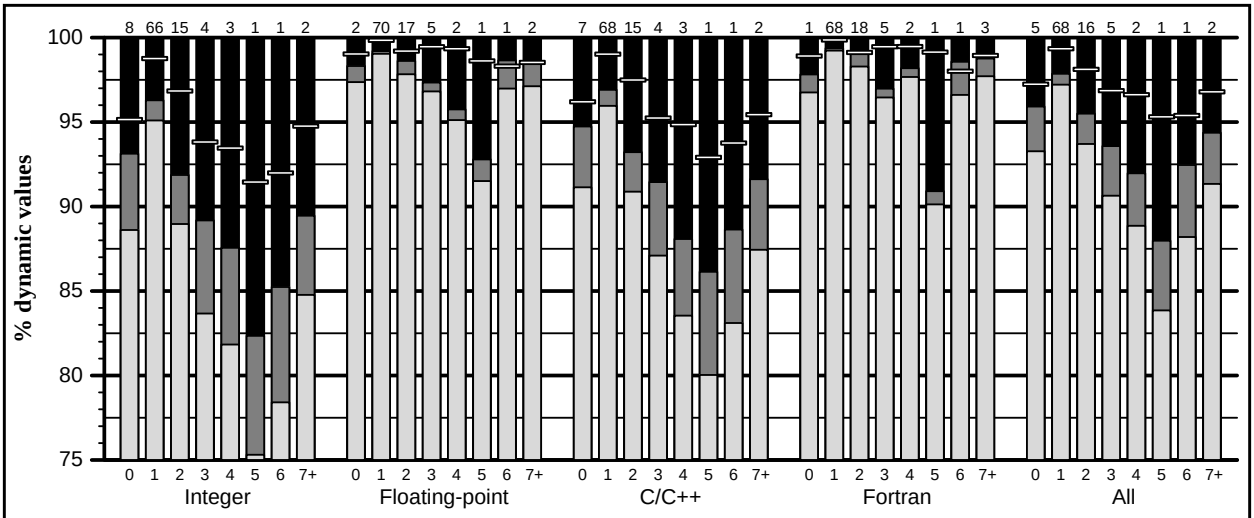


Figure 3.20. Predictor performance by predicted degree

ability of the control-flow signature contribute to the determination of their final degree of use. The coverage decrease that accompanies the drop in accuracy is a result of the confidence policy reacting to these more difficult-to-predict instructions. Coverage and accuracy bottom out at a predicted degree of use of five. The higher accuracy of the limiting degree category (7+) may be explained by its inclusion of multiple degrees of use and the absence of underpredictions. There is no apparent reason for the increased coverage and accuracy of predicted degrees of use of six versus five, but a similar anomaly at six uses was observed in Section 2.3.

3.4.6 Training

Any dynamic prediction technique depends on a means of observing the run-time behavior of the predicted property. These observations modify the stored state, training the predictor to recognize and predict subsequent occurrences of the same behavior. The dynamic observation of degree of use requires the ability to see and count all uses and definitions of values performed by the program, which involves tracking the source and destination registers of all executed instructions.

It is important to emphasize that degree of use applies to dynamic values—registers are only a means of naming particular dynamic values. An instruction that generates a value specifies a destination register, binding that register to the value and giving the value a name. The degree of use of the bound value increases as subsequent instructions name that value (via the register) as an input. Eventually, all uses of the value will occur, and, because registers are a limited resource, the associated register will be reclaimed by binding it to a new value. Once a new value is bound

to a register, the old value with which it was associated can no longer be named or used. Thus, the only way to dynamically detect that no more uses will occur is to observe the reclamation of its associated name.

In most superscalar processors, there are two register namespaces—architectural and physical. Degree of use may be computed using either, but there are two reasons to prefer using the architectural namespace. First, the architectural namespace is smaller. Therefore, the number of values that must be simultaneously tracked is smaller. Second, architectural registers have more convenient semantics with respect to their reuse. The creation of a new value in an architectural register implies the destruction of the prior value named by that register. The freeing of a physical register, however, is a separate event from its old value becoming un-nameable. Both of these events must be handled separately or the latency of the degree of use computation would be higher.

One other point must be made regarding computing degree of use for values in architectural registers. Because registers are reused, the value being named by a particular architectural register depends on when the name appears within the instruction stream of the program. A given physical register refers to only one value for all in-flight instructions, but many values may be simultaneously associated with a single architectural register. Therefore, the determination of degree of use must take place on an in-order instruction stream when tracking uses via architectural registers.

In an out-of-order processor, there are two possible in-order streams of instructions that may be considered: the fetch stream and the retirement stream. The fetch stream has the disadvantage of containing wrong-path instructions. Using this stream for training implies higher training bandwidth and the possibility of training the predictor with spurious use information. Also, the training state must be recovered upon branch mispredictions and other mis-speculations to avoid attributing uses to the wrong values. Training the predictor with the retirement instruction stream suffers none of these problems, but exhibits higher training latency. Because predictions are made early in the pipeline, waiting for the instructions to retire before adjusting the behavior of the predictor results in much lower responsiveness. Also, the control-flow information must still be captured early in the pipeline (where it is used for making predictions) and kept with the instruction until it reaches the training structure at retirement.

Figure 3.21 contrasts the two different training algorithms. The R bars represent the performance of a predictor trained using the retirement instruction stream, as has been assumed until now. The F bars show the performance when the predictor is trained from the fetch stream (actually the instruction stream just before renaming where the predictions are consumed). Training on the fetch stream carries a substantial accuracy penalty relative to the coverage benefit. Only for very low cost factors (less than 1.25 here) is the gain in coverage worthwhile. Therefore, use of the retirement instruction stream is preferred for training.

The actual mechanism for calculating the degree of use for training is straightforward. For each architectural register, one maintains a counter that is incremented when a use of the corresponding register is observed. The counters saturate at the maximum predictable degree of use, a limit discussed in Section 3.2.1. When a register is overwritten, the counter value equals the degree of use of the value previously in the register (subject to the limit). The counter is reset and the process resumes.

This set of counters is referred to as the *degree training table* (DTT). In addition to the counter, each entry contains information about the dynamic instruction that produced the value currently in the corresponding register (e.g., the static instruction address and control-flow signature). This information is used with the final degree of use in training the predictor. If the degree of use predictor is to be trained on the rename instruction stream, the DTT can be merged with the

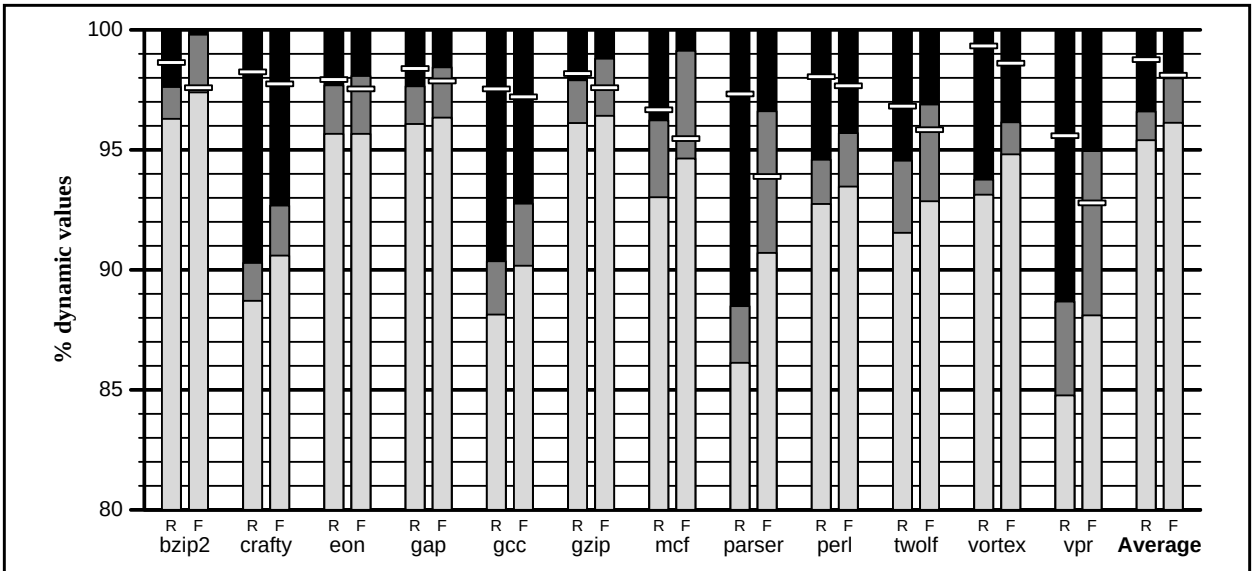


Figure 3.21. Predictor training with rename vs. retirement instruction streams

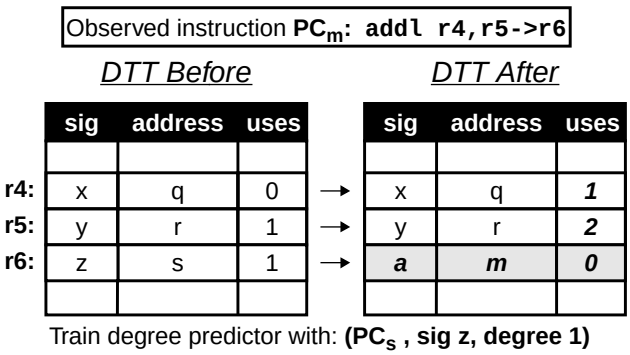


Figure 3.22. Structure and operation of a degree training table

rename instruction map, which also is indexed by architectural register and is recovered on mis-speculations.

The operation of the DTT is illustrated in Figure 3.22. As the instruction at PC_m is observed, it increments the use counters corresponding to its source registers. Because this instruction produces a value, it also updates the entry for the destination register (shaded). The new contents are created by resetting the use counter and setting the instruction identifying fields and the predicted degree of use appropriately. The prior contents of the written entry are forwarded to the predictor for training.

Once at the degree of use predictor, a write access is performed. A write miss always results in a replacement (refer to the discussion of replacement policies in Section 3.4.5). Write hits may be subdivided into *conflicts* and *confirmations*. Conflicts occur when the training and stored degrees of use differ while confirmations occur when they match. The actions taken in these two cases depend on the prediction algorithm. In the simple predictor, the training degree is written in both cases (i.e., the stored state does not actually change on a confirmation). In the predictors with confidence counters, the confidence is adjusted upwards on a confirmation and downwards on a conflict. A conflict when the confidence counter is zero results in a replacement. As described in Section 3.4.3, if the easy bit is set in a predictor using this optimization, the conflict causes replacement of the stored degree of use regardless of the value of the confidence counter.

3.4.7 Verifying degree of use predictions

The detection of degree of use mispredictions is closely related to the training of the predictor. In both cases, an instruction stream is observed and the degree of use for all live values determined.

Underpredictions are detected as soon as the first use beyond the expected number is observed. The detection of an overprediction must wait until the final degree of use is known to be less than that predicted. As for the training process, there is a choice of which instruction stream is used to detect mispredictions. Detecting mispredictions on the fetch stream gives lower misprediction detection latency at the cost of some false misprediction signals due to wrong-path execution. The verification latency ultimately determines the amount of state that must be maintained in speculative applications of degree of use prediction. Also, long verification latencies imply long misprediction recovery times, increasing the cost of mispredictions.

The exact verification latency depends on characteristics of both the program and the processor. The number of intervening instructions between consecutive definitions of the same register is a property of the workload (see Figure 2.4); the time taken to process these instructions, however, depends on the details of the microarchitecture, as does the rate of spurious mispredictions. Any measurement of these properties would be specific to the workload and pipeline configuration (e.g., in the original work on degree of use prediction [14]), not to the method of degree of use verification, so these data are omitted here.

The structure performing misprediction detection is called the verification table or VT, and it is conceptually similar to the DTT. Where the DTT keeps an address and other information to associate with the dynamically-observed degree of use, the VT keeps the predicted degree of use for comparison purposes. The VT must also keep information to identify each instruction with a pending prediction. When a misprediction is detected, this information is forwarded to the users of the predictions to initiate any necessary recovery actions. Thus, the nature of the identifying information stored in the VT depends on how the degree of use predictions are being used. If both training and misprediction detection occur at the same point within the pipeline, these two structures can be merged.

3.4.8 Predictor bandwidth

Another consideration in the design of the predictor storage structure is the required access bandwidth. The predictor must be read for each instruction renamed and written for each instruction renamed or retired (depending on the training method). Normally, each separate simultaneous access to a storage structure requires a dedicated port. As both area and access time of a storage

structure scale up with the number of ports, the cost of supplying the predictor's read and write bandwidth could be prohibitive for a wide superscalar machine. Fortunately, blocking and/or banking can be used to supply the necessary bandwidth with fewer access ports.

Blocking leverages the spatial locality inherent in consecutive read accesses to the degree of use predictor. Because the degree of use predictor supplies predictions to instructions in the fetch order, accesses to the degree of use predictor have the same spatial locality as do the instructions in the execution stream itself. Rather than supplying access ports to the predictor that match the peak fetch bandwidth of the machine, the state for multiple consecutive instructions can be fetched with a single predictor access. An additional advantage is that the tag and control-flow signature can be shared among all of the degrees of use in a block (confidence information and must still be maintained per-instruction).

Blocking does have some significant disadvantages, however. Balancing the decrease in storage due to sharing tags and control-flow state is an increase in predictor storage from wasted entries. As not all instructions produce a value, providing predictor state to multiple consecutive instructions introduces the likelihood of allocating predictor entries to instructions for which degree of use is meaningless. Also, the determination of the degrees of use of values generated by consecutive instructions are likely to occur across many different cycles. Thus, blocking does not really help reduce the write bandwidth requirement.

Another means to deal with the access bandwidth requirements of the predictor storage is banking. Again, the spatial locality of predictor read accesses comes into play, suggesting that bank conflicts will not be a significant problem. In the absence of taken control instructions, read accesses in a given cycle are to consecutive instruction addresses, which will reside in different banks. Therefore, banking the predictor to the same degree as the fetch width (i.e., eight banks in an eight-wide machine) is sufficient.

The lack of locality in write accesses does not pose as much problem for a banked predictor as for one with a larger block size. Although bank conflicts will be more likely for write operations, the average write bandwidth can be sustained. Assuming write accesses are evenly distributed across n single-ported banks and a full complement of n writes is attempted, the expected average write bandwidth attained will be at least $0.64n$ for $n \leq 23$. Since the average fetch bandwidth is less than the fetch width and training is only required for the 80% of instructions that produce val-

ues (see Table 2.1), banking can provide adequate average write bandwidth. Queuing write operations between the DTT and the predictor can reduce the incidence of dropped writes due to bank conflicts.

3.5 Hybrid Prediction Schemes

Hybrid degree of use prediction schemes combine elements of static and dynamic predictors to exceed the performance of either in isolation. A static predictor can offer perfect accuracy, but only on some instructions. Dynamic predictors offer much better coverage, even handling instructions with variable behaviors, but at the cost of hardware resources and lower accuracy. Strategies for hybrid prediction differ based on what static information is available and how that information is used. Given the high accuracy and coverage demonstrated by reasonably-sized dynamic predictors, there is likely to be little need for complex hybrid schemes. Also, there is a tremendous design space of hybrid predictors, the exploration of which is outside of the scope of this work.

However, one very simple hybrid prediction model may be worth considering where static degree of use information is available. This scheme uses static predictions where analysis identifies a single unique degree of use for a static instruction and employs dynamic prediction for the remaining instructions. Such a combination would yield improved coverage over a dynamic predictor (or equivalent coverage with less hardware) since the hybrid predictor would not need to maintain state for those instructions with available static predictions (the same instructions that motivated the introduction of the easy-bit optimization in Section 3.4.3). Also, static predictions can be supplied immediately without a delay for the training of the dynamic predictor. Accuracy improves due to the reduction in aliasing within the dynamic predictor and the perfect accuracy offered by static analysis.

3.6 Summary

Degree of use prediction offers a practical method for exploiting the knowledge that degree of use provides about values. The degree of use of a value cannot be known until all of the uses of the value have occurred *and* the register holding the value has been reclaimed. At this point, the applicability of degree of use knowledge is questionable since the associated value may already

have been created and distributed to its consumers. Degree of use prediction affords knowledge about a value before the value even exists. This knowledge can therefore be used to guide the allocation of microarchitectural resources, the value communication method, and the handling of the instructions generating and consuming that value.

Degree of use is a property of the program's dataflow structure making it amenable to static dataflow analysis. Although interprocedural analysis is required, the necessary dataflow equations are straightforward and can be solved with well-understood techniques. Applying this analysis on the SPEC CPU 2000 benchmarks reveals that over 60% of the static instructions always generate values with a unique, statically-identifiable degree of use. Applying profiling information enables static prediction accuracies approaching 99% on over 85% of dynamic instruction instances.

Accurate dynamic degree of use prediction is also possible. Degree of use exhibits considerable locality with respect to individual static instructions. Most static instructions generate values with the same degree of use during every execution. When instructions can produce values with multiple degrees of use, values produced consecutively are still likely to have the same degree of use. Therefore, history-based prediction methods that predict future behavior based on on-line observation are very successful. Simply predicting the last-observed degree of use for each static instruction is good for 95% accuracy with perfect coverage. By employing confidence counters and control-flow information, more sophisticated dynamic prediction algorithms can deliver higher accuracies at the cost of lower coverage.

The best dynamic predictor presented in this chapter offers 98.8% average accuracy at 96.6% coverage with 13.4K-bytes of storage. This level of performance is enabled through the use of future control-flow information to distinguish instances of the same static instruction with different degrees of use. Future control-flow information is available because of pipelining: control-flow predictions for instructions in a pipeline indicate the future path of execution with respect to instructions later in the pipeline. Originally put forth in the work on useless instruction elimination [15], the description and exploitation of future control-flow represents a novel contribution of this work with applications beyond degree of use prediction (e.g., its recent application to branch prediction [31]).

Chapter 4

Useless Instruction Elimination

The data of Section 2.1 indicate the existence of a non-negligible number of instructions with a degree of use of zero, especially among the integer benchmarks. These correspond to dynamic instances of value-generating instructions whose results are not required by the program. In terms of value communication, these instructions represent the degenerate case of non-communication. In the absence of other side-effects resulting from these instructions, the behavior of the program is completely unaffected by the execution (or non-execution) of these instructions. The performance, however, can be negatively impacted when these instructions cause contention for processor resources. Even when they do not delay more useful work, such instructions represent wasted effort, reducing a processor's efficiency.

Zero-use dynamic values frequently arise from static instructions that can produce non-zero degrees of use as well. Thus, it is important to differentiate between static and dynamic instructions when referring to instructions throughout this chapter. The term *useless instruction* is introduced to refer to a dynamic instruction that has no consumers (i.e., a result with degree of use zero); conversely, dynamic instructions having consumers are *useful instructions*. All dynamic instructions fall into one of these two categories. The taxonomy of static instructions is more complex. Static instructions incapable of having useful instances are *dead instructions*; ideally, these are detected and eliminated by the compiler during dataflow analysis and optimization. Static instructions that are capable of generating useful instances, even when no such instances

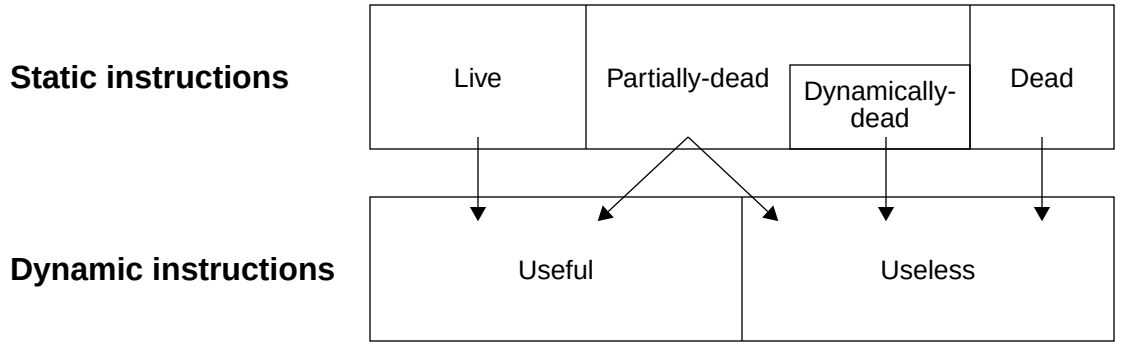


Figure 4.1. Instruction taxonomy

occur in a particular execution, are called *partially-dead* instructions [53]. That subset of partially-dead instructions that generate only useless instances in a given execution are referred to as *dynamically-dead instructions*.† *Live instructions*, which always generate useful instances, comprise the remaining static instructions. The taxonomy of static and dynamic instructions described by these definitions is illustrated in Figure 4.1 The first part of this chapter investigates the prevalence and properties of useless instructions from these different sources.

This chapter then develops *useless instruction elimination*, which is a mechanism to avoid the register allocation, scheduling, and execution of useless instructions identified through degree of use prediction. Eliminated instructions are kept in a dedicated structure until their status can be verified by the retirement of the instruction that renders them useless (i.e., the instruction that overwrites the same architectural register defined by the useless instruction). The retirement of instructions following the elimination candidate is prevented until verification occurs. Successfully-eliminated instructions can lead to reductions in resource utilization covering physical register management (allocation and freeing), register file read and write traffic, register file and load queue occupancy, and data cache read bandwidth. Small performance improvements are also possible in resource-constrained architectures.

The next section characterizes useless instructions in detail, looking at their origins, prevalence, and relationship to compilation. Section 4.2 describes a mechanism for useless instruction

† The terminology used here is different from that used in the original work on this topic [15]. In that work, “statically-dead instructions” included both dead and dynamically-dead instructions, while “useless instruction” and “dynamically-dead instruction” were used interchangeably.

elimination, which is subsequently evaluated in Section 4.3. Related work is presented in Section 4.4, and Section 4.5 summarizes the chapter.

4.1 Characterizing Useless Instructions

The potential benefit of useless instruction elimination depends on the prevalence of these instructions during actual execution. Zero-use values exhibit more variability in the frequency of their occurrence among the different benchmarks and compilation environments versus values with other degrees of use (as shown in Figure 2.1). In this section, the sources of useless instructions are identified and investigated in order to understand why they exist and what factors contribute to their prevalence.

4.1.1 Origin

Figure 4.2 presents four examples of assembly code responsible for useless instructions. Each code fragment was extracted from an optimized version of the indicated benchmark (the tuned configuration used for the studies in Chapter 2 and described in Section A.2.2 of the appendix). The dead or partially-dead instructions that generate useless instances are highlighted with the destination register in boldface. Subsequent references to the register are also in boldface, and overwrites are circled. Possible paths of control flow are indicated with arrows. Ellipsis points indicate the omission of unrelated code.

Figure 4.2(a) shows two dead instructions of the simplest possible kind—there are no control instructions between the creation of the dead values and their subsequent destruction shortly thereafter. Very few actual useless instructions result from such constructs because this situation can easily be identified by dead code elimination. Even peephole optimization can detect this pattern when the uses and definitions belong to the same basic block (as is the case with the second dead instruction and its overwrite; the first dead instruction belongs to a different basic block since the second is a branch target).

Useless instructions arise more frequently from code like that in Figure 4.2(b), which shows a partially-dead instruction. In this case, a value is placed into a register `t0` immediately prior to a conditional branch. On the fall-through path of the branch, the value is overwritten; if the branch is taken, however, the value is useful. The partially-dead instruction in this example also happens

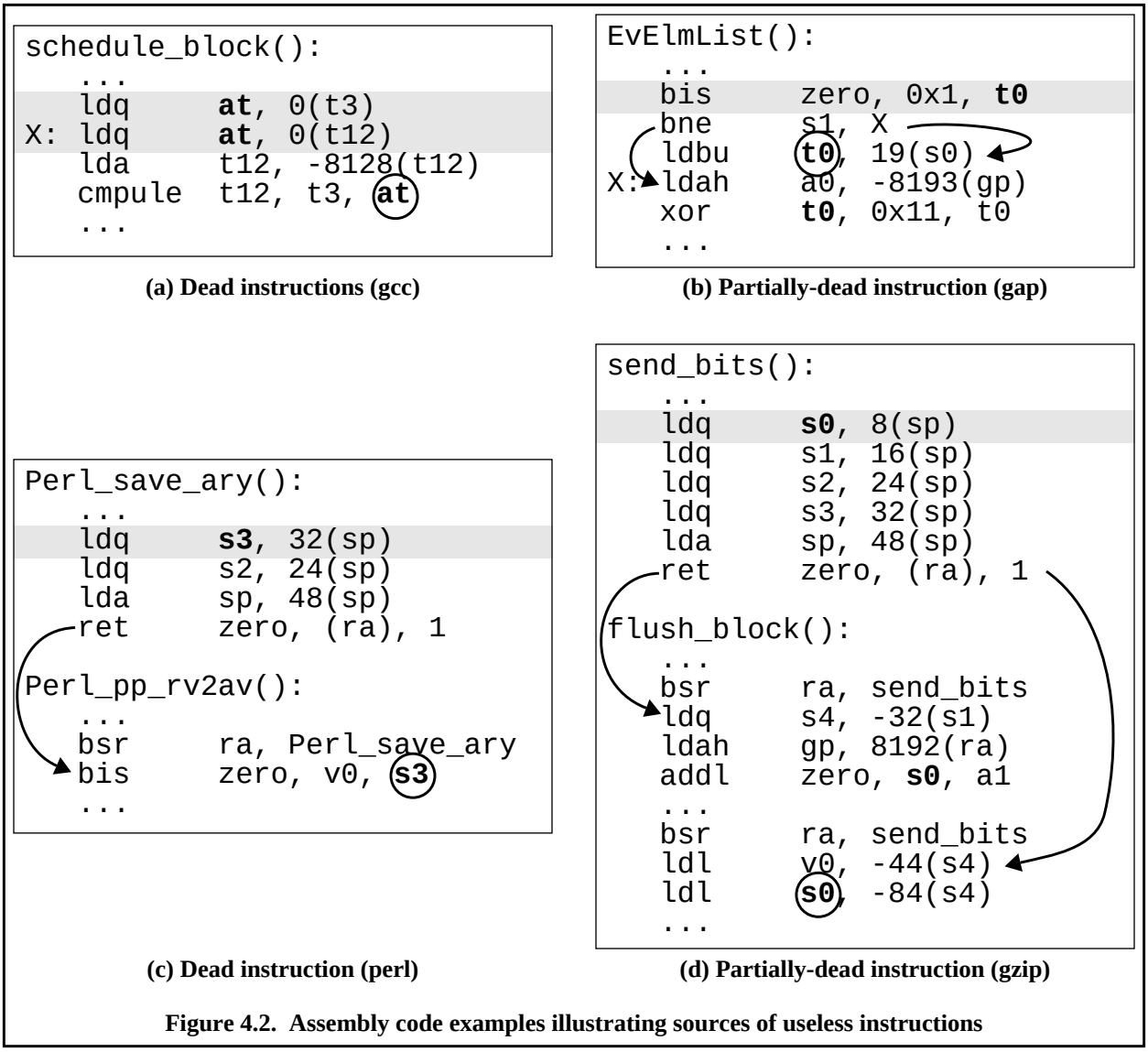


Figure 4.2. Assembly code examples illustrating sources of useless instructions

to be dynamically-dead (i.e., the branch was never taken during the execution of the benchmark with this particular set of inputs). Dynamically-dead instructions, including this one, are often associated with code for the detection of run-time errors.

Interprocedural control-flow can also be responsible for dead and partially-dead instructions. The first `ldq` of Figure 4.2(c) restores the value of the callee-saved register `s3` prior to returning to the calling procedure. However, the value in `s3` was dead prior to the procedure call as evidenced by its immediate overwrite afterwards. Since `Perl_save_ary()` is only called from this single call site, the highlighted instruction is a dead instruction. Note that the corresponding register save at the beginning of `Perl_save_ary()` (not shown) is also unnecessary, but not

useless (since it modifies memory). Figure 4.2(d) shows a more typical case where the restored value is live at some call sites and dead at others (only one of each is shown).

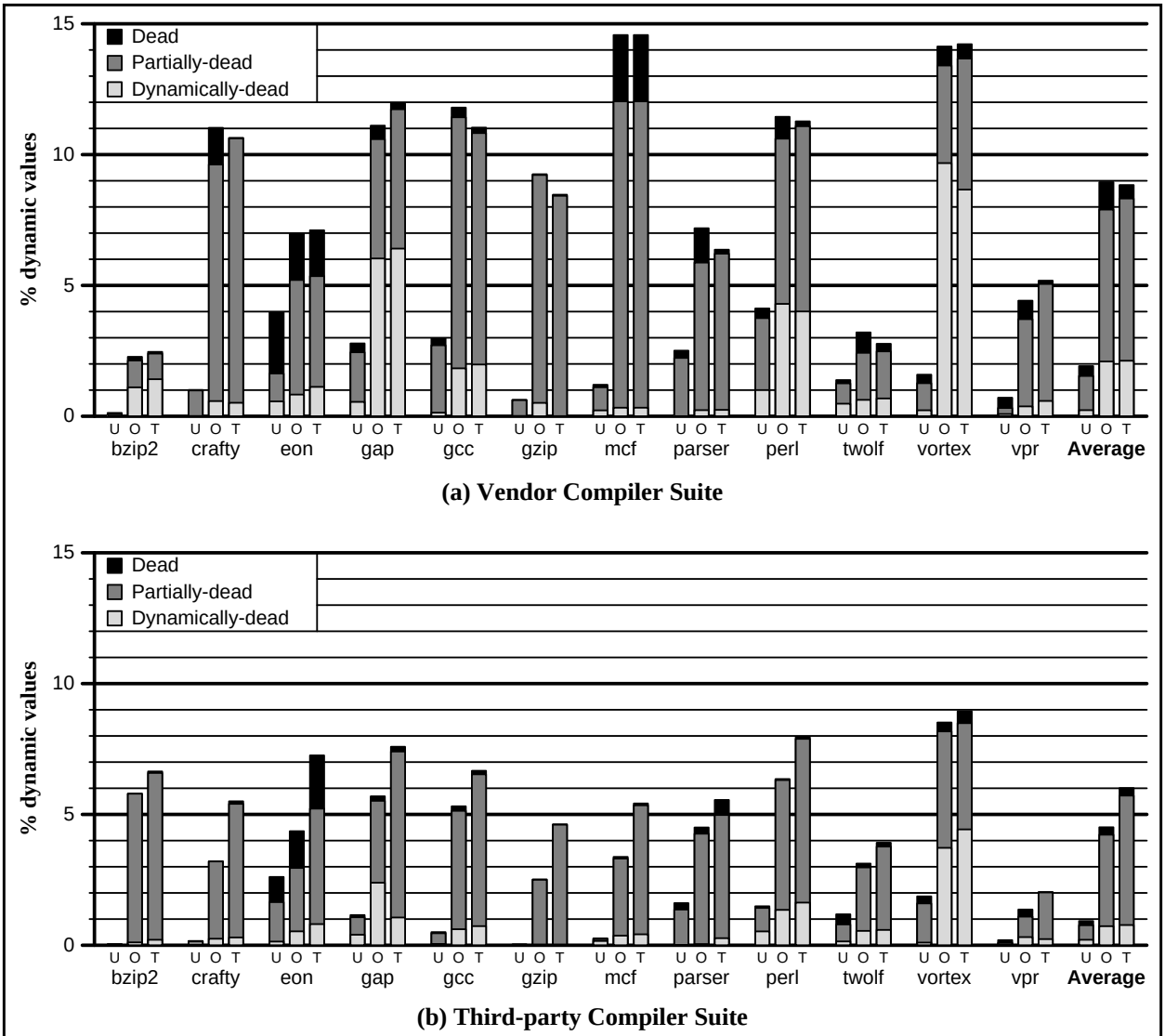
The code examples of Figure 4.2 demonstrate the extent to which the compiler affects the incidence of useless instructions. Ideally, compiler optimization should leave no dead instructions. The situation with respect to partially-dead instructions is more complicated. Just because the compiler creates or allows partially-dead instructions to exist does not imply that there are not good reasons for doing so. Section 4.1.3 discusses some of the reasons why partially-dead instructions are likely to continue to exist regardless of the sophistication of the compiler.

4.1.2 Prevalence

The characterization data in Chapter 2 demonstrated that different compilers generate code with different amounts of useless instructions. The influence a compiler could have was also evident in examples presented in Figure 4.2. However, the compiler does not generate useless instructions *per se*; rather, the compiler creates the dead and partially-dead instructions that lead to useless instructions during execution. Examining the prevalence of the useless instructions themselves tells of the potential and properties of useless instruction elimination, which operates dynamically. The incidence of the static instructions that lead to these useless instances, however, illuminates the role of the compiler in their occurrence. Therefore, in this section, both the static and dynamic aspects of useless instructions will be considered.

Figure 4.3 shows the incidence of useless instructions for three different compilations of each benchmark on each of two different compilers. The floating-point benchmarks are omitted here and throughout this chapter due to the almost negligible quantity of useless instructions that they exhibit. The bars show the contributions of useless instructions by dead, partially-dead, and dynamically-dead instructions as percentages of all dynamic instructions. The `U` bars represent the unoptimized benchmarks (`-O0`), the `O` bars the lightly-optimized benchmarks (`-O1`), and the `T` bars the tuned (i.e., highly-optimized) benchmarks. Details on the compilers and the full command line options for each configuration can be found in Section A.2 of the appendix.

The increased incidence of useless instructions among the optimized benchmarks is striking. An average of 2% of the dynamic instructions in the unoptimized benchmarks are useless; in the optimized benchmarks, the figure is nearly 9% (vendor compiler suite). Each benchmark exe-



(a) Vendor Compiler Suite

(b) Third-party Compiler Suite

Figure 4.3. Prevalence of useless instructions

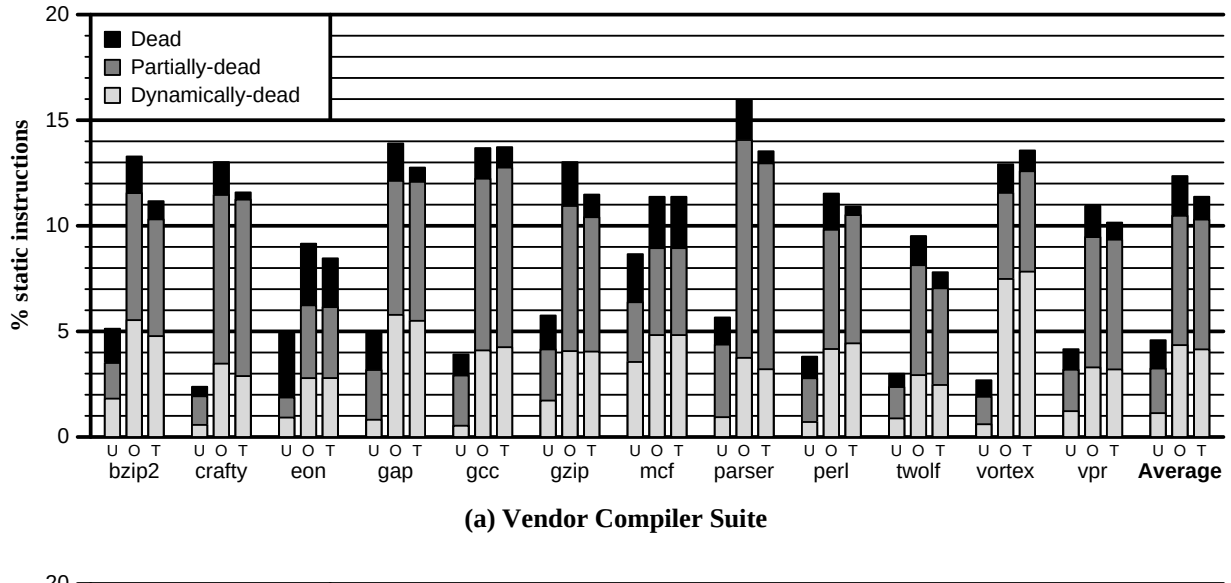
cutes a different number of instructions under different compilation options. As a result, a higher fraction of useless instructions does not necessarily imply that the absolute number of them has increased. However, the number of useless instruction instances is *also* increased by an average factor of 3.1–3.9 in the optimized and tuned configurations for both compilers.

Only `eon` exhibits a reduction in the absolute number of useless instructions when compiled with optimization (again, under both compilers). This benchmark is also the only one that exhibits a significant fraction of dead instructions in the unoptimized binaries. Examining the ten static instructions responsible for the most useless instances (32% altogether) in the unoptimized version (from the vendor compiler) explains these phenomena. Of these ten instructions, eight are

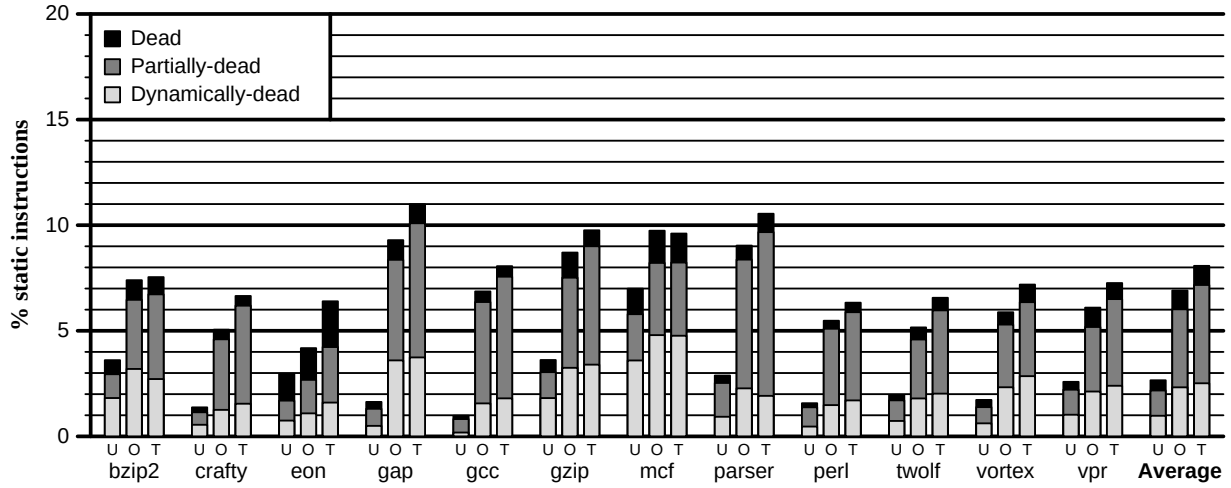
dead (a ninth is dynamically-dead). All ten of them occur soon before (i.e., with no intervening control instructions) a procedure return. Nine of them generate the return value of a simple class member function, and in all but one of these, the return value is the implicit return value of a class constructor (i.e., `&this`). None of the objects involved are base classes, so all calls to these simple member functions are easily identified and replaced with inlined versions of the functions. Inlining exposes the "deadness" of the return values to intraprocedural dead code elimination. As a result, the absolute number of dead instructions drops some 73% when the lowest level of compiler optimization is applied. Since dead instructions cause such a large portion of the total useless instructions, the absolute number of useless instructions is reduced. However, the accompanying reduction in overall instruction count ensures that `eon`, like every other benchmark, generates a higher fraction of useless instructions when compiled using optimization.

In every benchmark, partially-dead instructions are almost entirely responsible for the increase in useless instructions with optimization. The compiler does not, in general, influence the broad execution characteristics of the program. For example, the nature of accesses to a data structure will be invariant with respect to compilation, irrespective of such optimizations as loop unrolling or function inlining. Therefore, the increased incidence of useless instructions from partially-dead sources after optimization should result from an increase in the number of these static instructions and not from a sudden increase in the frequencies of their execution. Figure 4.4, which provides a breakdown of the number of static instructions, confirms that this is the case. However, comparing Figure 4.4 with Figure 4.3 shows that the relative increase (with optimization) in the amount of dead and partially-dead static instructions is less than the relative increase in the number of useless instructions from these sources. Thus, the new partially-dead instructions generate comparatively more useless instances than the ones that existed prior to optimization. This effect is related to why partially-dead instructions arise from compiler optimization and is explored in Section 4.1.3.

The difference in incidence of useless instructions between the two compilers is small relative to the change seen with the enabling of optimization. The types of analyses that the compilers perform are similar as are the end results in terms of the kinds of dead and partially-dead instructions left after compilation. The main difference between the two different compiler suites is that the benchmarks compiled with the vendor compilers universally exhibit more useless instructions.



(a) Vendor Compiler Suite



(b) Third-party Compiler Suite

Figure 4.4. Prevalence of static instructions contributing useless instances

At the same time, these benchmarks perform better than those compiled with the third-party compilers at similar levels of optimization. For the remainder of the chapter, only the tuned benchmarks compiled with the vendor compilers will be used.

Transitively-useless instructions generate results used only by useless instructions or other transitively-useless instructions. The incidence of transitively-useless instructions is about 33% that of useless instructions [15]. Their relatively small occurrence is due to the short average length of register dependence chains [50] and the likelihood that a useless instruction has no data-flow predecessors (e.g., due to a higher fraction of load-immediates among useless instructions

than among all instructions). Following dependence chains through memory allows yet more instructions to be established as transitively-useless [71], but register degree of use prediction cannot help to identify any of them. Also, it is significantly more difficult to verify dynamically that an instruction is transitively useless: *all* values derived from any dependent instruction must be overwritten prior to their use by any instructions that are not themselves dependent. Because of the increased complexity of detecting and verifying transitively-useless instructions and their limited number, they are not considered further.

4.1.3 Role of the compiler

To delve further into the role of compiler optimization, note that compilation occurs in two stages: translation of the source program followed by optimization. An unoptimized binary corresponds roughly to a simple translation of the source code into machine language. A dead or partially-dead instruction then can be said to be *pre-existing* with respect to the original program if it exists in the unoptimized binary. For example, a variable set before an `if` statement and used only if the associated condition were true would result in a partially-dead instruction in the unoptimized binary. Alternatively, a dead or partially-dead instruction created by the optimization process itself is *generated*.

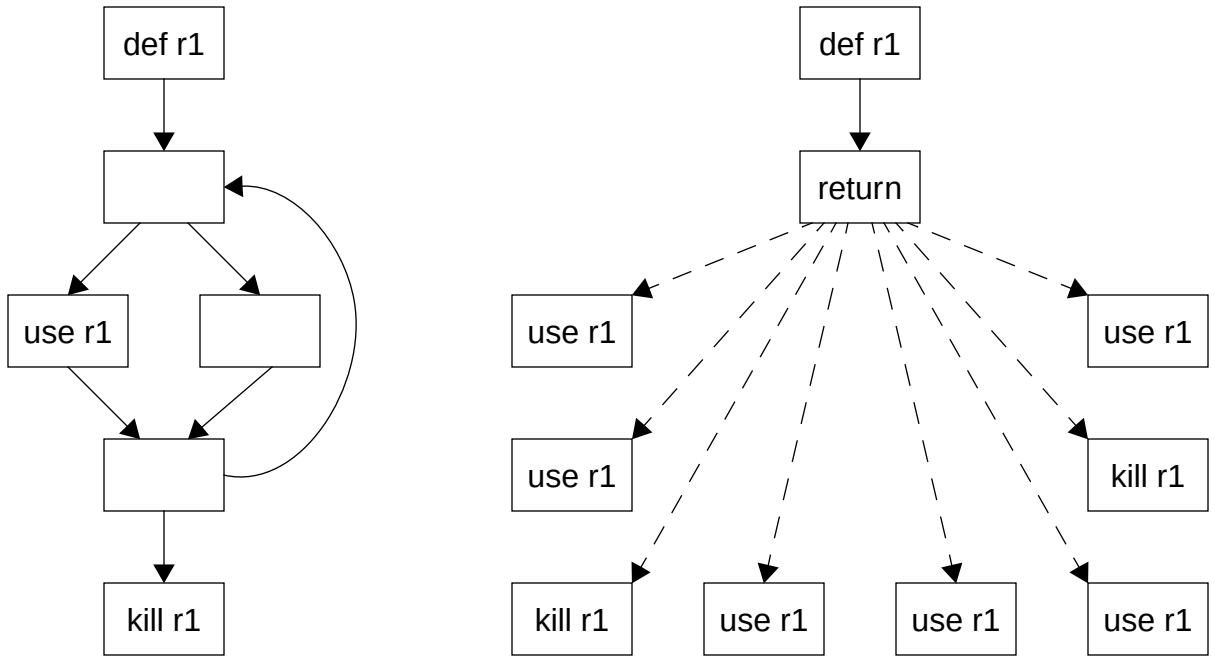
Dead instructions of either class should *never* remain in a program after optimization. Those that survive optimization do so by evading detection—usually by virtue of generating values that cross procedure boundaries. Some of these are pre-existing; others are generated by compiler optimizations, many of which produce dead instructions as a normal side effect. The vast majority of these are eliminated during a later optimization phase, but some can slip through. In `mcf`, for example, two particular instructions in the unoptimized code have their only consumers removed during optimization, but cannot be eliminated because their results are thought (incorrectly) to be used in a procedure called several instructions later. These two instructions are responsible for 90% of the post-optimization useless instructions attributed to dead instructions.

Just as was the case for dead instructions, pre-existing partially-dead instructions may evade detection. The interprocedural partially-dead instruction of Figure 4.2(d) is likely such a case. Assuming they can be detected, however, a specific compiler optimization—partial dead code

elimination—exists to remove them [13]. This algorithm attempts to move partially-dead code down into the specific paths where it is live.

Unfortunately, the elimination of some partially-dead instructions carries a cost that makes this operation undesirable. Figure 4.5 illustrates two such cases as portions of simplified control-flow graphs. The partially-dead instructions (def) generate values that are used (use) or overwritten first (kill) depending on the flow of control. In the example of Figure 4.5(a), the partially-dead instruction generates a value possibly used within a subsequent loop body. Moving the definition to just before its consumer can result in a significant execution time penalty if the loop is executed many times. Elimination of the partially-dead instruction of Figure 4.5(b) increases code size due to the degree of duplication required to ensure it ends up on each path where it is live. Procedure cloning [23], which generates specialized copies of entire procedures, may be required to eliminate all of the partially-dead instructions in similar cases.

While difficulties with detection and costs of removal explain why pre-existing partially dead instructions may not be significantly reduced by optimization (assuming that the compiler is even capable of these optimizations), it does not account for the new partially-dead instructions intro-



(a) Increased execution time **(b) Increased code size**

Figure 4.5. Pitfalls of eliminating partially-dead instructions statically

duced by optimization. One can surmise that the partially-dead instructions that are introduced with optimizations involve intraprocedural control flow. Other than inlining, compilers will not generally move instructions across procedure boundaries, especially at the lowest level of optimization represented in Figure 4.3. Therefore, the predominant cause of new partially-dead instructions must be the movement of instructions above one or more branches that determine their liveness, and the new partially-dead instructions will resemble the one in Figure 4.2(b) rather than Figure 4.2(d).

Compiler optimizations that can move instructions across basic block boundaries include loop-invariant code motion and code scheduling. Loop-invariant code motion moves instructions that compute a loop-invariant value outside of the body of the loop. Essentially, it is the reverse of the partial dead code elimination transform. During code scheduling, the compilers move computations across branches to facilitate better performance on the expected target architecture [20, 22]. For example, initiating a long-latency operation earlier than when it is guaranteed to be used may reduce execution time along the path(s) on which it is used. Compilers for statically-scheduled processors must move instructions to account for resource constraints and execution latency. For example, a compare that computes the condition for a branch may need to be moved to an earlier basic block if enough instructions cannot be found to cover its latency in the block containing the branch.

The creation of partially-dead instructions by genuinely beneficial compiler optimizations suggests that optimization will always generate new partially-dead instructions. In addition, the inability to detect pre-existing dead and partially-dead instructions and the high cost of eliminating some partially-dead instructions ensures that these will continue to remain in optimized code as well. Therefore, it is expected that useless instructions will continue to exist in significant quantities in future optimized programs, especially where the control-flow (both intra- and interprocedurally) is complex. To the extent that these instructions consume scarce resources, they will negatively impact performance.

4.1.4 Useless instruction resources

The resource reduction potential of useless instruction elimination depends not only on the incidence of these instructions but on the types of resources that they normally require. Every useless

instruction eliminated saves a physical register—for the unneeded result value—and reduces register file write bandwidth. However, the elimination of a useless load can also reduce cache bandwidth and load queue occupancy.

Table 4.1 presents data on the nature of the useless instructions with respect to their resource requirements. For each class of instructions listed across the top of the table, the top number reflects the percentage of useless instructions in the class, while the bottom number (shaded) represents the percentage of overall value-generating instructions in the class. Integer register moves, load-immediates, and load-addresses are separated from other integer operations because of the large differences in their representation in useless instructions versus other instructions. However, all four of these classes (load-address, load-immediate, move, and integer ALU operations) of instructions require an integer ALU for execution. Some instructions may belong to different classes depending on the inputs. For example, a `bis` (logical OR) instruction can be a move (`zero` register, other register), a load-immediate (`zero` register, immediate or two `zero`s), or an ALU operation (all other cases). Control instructions, which cannot be eliminated due to their side effect of setting the program counter, are not represented among useless instructions.

The occurrence of useless loads is of great interest because of the extra resources required by these instructions. In addition to the resources consumed by any ALU instruction (e.g., a destination register), loads also require load queue entries and cache bandwidth. Loads are well-represented among useless instructions, accounting for an average of 30% of them, which is nearly equal to their average incidence among all value-generating instructions. In spite of this overall parity, individual benchmarks exhibit significantly different ratios of loads to other operations among useless instructions than among all value-generating instructions. In some cases (most notably `twolf`), the useless instructions are “enriched” in loads, while in others (e.g., `mcf`), relatively few loads are useless. Benchmarks in the former category are likely to see larger relative reductions in cache bandwidth under useless instruction elimination.

Nearly all useless operations that are not loads are integer ALU operations (excepting a small fraction of useless floating-point operations, especially in `eon` and `vpr`). However, the kinds of integer ALU operations among useless instructions is significantly different than in value-generating instructions overall. A useless instruction is more than twice as likely to be a load-immediate,

Table 4.1: Types of Useless Instructions

Benchmark	Load (memory)	Load immed.	Load address	Integer reg. move	Integer ALU op.	Floating-point op.	Control
bzip2	34.82	13.77	25.86	12.68	12.86	0.00	0.00
	30.02	0.66	5.79	3.97	59.21	0.00	0.35
crafty	34.43	8.93	29.93	2.73	23.98	0.00	0.00
	28.31	3.09	20.25	2.16	44.59	0.00	1.59
eon	42.31	5.42	30.05	7.24	4.63	10.36	0.00
	36.90	3.45	18.06	4.73	13.56	20.03	3.19
gap	19.32	31.36	31.28	10.20	7.66	0.17	0.00
	31.89	5.90	26.25	6.40	26.98	0.21	2.37
gcc	31.45	23.64	12.29	13.57	19.02	0.02	0.00
	32.90	4.53	13.67	5.84	41.59	0.02	1.43
gzip	22.92	10.62	39.12	2.70	24.64	0.00	0.00
	24.20	5.82	10.57	0.47	58.15	0.00	0.79
mcf	16.81	2.46	33.66	37.84	9.21	0.02	0.00
	35.19	2.96	13.76	16.38	27.80	0.00	3.91
parser	27.65	9.71	10.10	33.24	18.99	0.31	0.00
	30.21	2.94	8.90	8.58	46.76	0.03	2.57
perl	28.96	21.52	17.38	17.24	13.93	0.96	0.00
	38.80	5.34	19.18	6.66	27.22	0.32	2.47
twolf	42.48	9.40	16.70	9.43	21.25	0.75	0.00
	30.94	2.38	8.92	3.53	45.96	7.28	0.99
vortex	26.35	36.33	13.09	12.88	11.20	0.13	0.00
	36.20	13.54	14.05	8.54	24.16	0.65	2.77
vpr	34.50	15.63	7.52	11.57	26.36	4.42	0.00
	37.19	1.92	7.20	3.93	39.00	9.58	1.18
Average	30.17	15.73	22.25	14.28	16.14	1.43	0.00
	32.73	4.38	13.88	5.93	37.92	3.18	1.97

a load-address, or a register move. More typical integer computations—adds and shifts, for example—occur less often among useless instructions. Recalling the origins of partially-dead instructions discussed in Section 4.1.3 helps to explain this phenomenon. Load-immediate and (frequently) load-address instructions create loop-invariant values. Thus, these instructions are

often candidates for hoisting outside of a loop body to reduce the loop's overhead. Sometimes this code motion will result in a new source of useless instructions. Register moves are most frequently needed to put a value into a specific architectural register before a procedure call or return. Values in the return-value and argument registers are often unused by the subsequent execution context, rendering the register move instructions useless.

In addition to the type of a useless instruction, the number of register inputs it requires is also important because it determines the demand placed on register file read bandwidth. Table 4.2 categorizes useless instructions by the number of register inputs. The actual number of non-`zero` register inputs is used; thus, an `addq` instruction with inputs consisting of an immediate and the `zero` register is counted as a zero-input instruction. The format of the table is identical to that of Table 4.1 except that the average number of inputs is presented in addition to the percentages for each category. Useless instructions exhibit an average of 23% fewer register inputs than value-generating instructions in general. Two-input instructions are significantly under-represented among the useless instructions while zero-input instructions tend to occur more often. These characteristics can be attributed to the higher incidence of operations such as load-immediates and load-addresses, which have zero and one input, respectively.

4.2 Useless Instruction Elimination

Exploiting the existence of useless instructions entails recovering the resources consumed by handling them. Obviously, these resources include the functional units used for execution, but they also include storage space for useless results (i.e., physical registers), cache bandwidth (in the case of useless loads), and issue bandwidth. Useless instruction elimination is a mechanism to filter useless instructions from the instruction stream early enough to avoid many of the overheads associated with them.

A high-level view of the integration of useless instruction elimination with a typical out-of-order pipeline is diagrammed in Figure 4.6. A degree of use predictor identifies candidates for elimination. After renaming their source operands, the candidate instructions are kept in a new structure (the *predicted-useless table* or PUT) until the speculation is verified. Renaming the source operands at elimination time significantly simplifies mis-speculation recovery; the only cost is the lost opportunity for reducing utilization of the rename structures. No destination phys-

Table 4.2: Number of Inputs of Useless Instructions

Benchmark	0-input	1-input	2-input	3-input	Average
bzip2	13.77	78.35	7.88	0.00	0.94
	1.01	57.51	39.74	1.73	1.42
crafty	8.93	76.59	13.00	1.47	1.07
	4.68	67.46	25.73	2.13	1.25
eon	6.16	84.32	9.52	0.00	1.03
	5.94	70.03	23.25	0.78	1.19
gap	31.37	63.30	5.28	0.05	0.74
	6.55	76.71	16.53	0.22	1.10
gcc	23.65	66.18	10.11	0.05	0.87
	5.95	74.34	19.13	0.59	1.14
gzip	10.62	83.74	5.64	0.01	0.95
	6.62	57.27	35.69	0.41	1.30
mcf	2.46	96.79	0.74	0.01	0.98
	6.86	72.94	19.95	0.25	1.14
parser	9.73	80.31	9.95	0.00	1.00
	5.52	63.61	25.67	5.20	1.31
perl	21.52	72.70	5.65	0.12	0.84
	6.97	77.68	14.50	0.84	1.09
twolf	9.40	78.35	7.88	0.00	1.03
	1.01	57.51	39.74	1.73	1.31
vortex	36.34	57.55	1.60	4.51	0.74
	16.39	72.10	10.77	0.74	0.96
vpr	15.63	67.52	16.83	0.02	1.01
	3.10	64.15	30.91	1.84	1.31
Average	15.80	75.49	8.19	0.52	0.93
	6.08	68.31	24.04	1.57	1.21

ical register is allocated (although one may be reserved; see Section 4.2.7). After the instruction's degree of use is confirmed to be zero, the PUT entry may be reclaimed and the instruction dropped without any effect on the correctness of the program's execution. After describing this mechanism in more detail, specific issues surrounding retirement back-pressure, misprediction recovery, the handling of loads and instructions with side effects, and deadlock avoidance will be

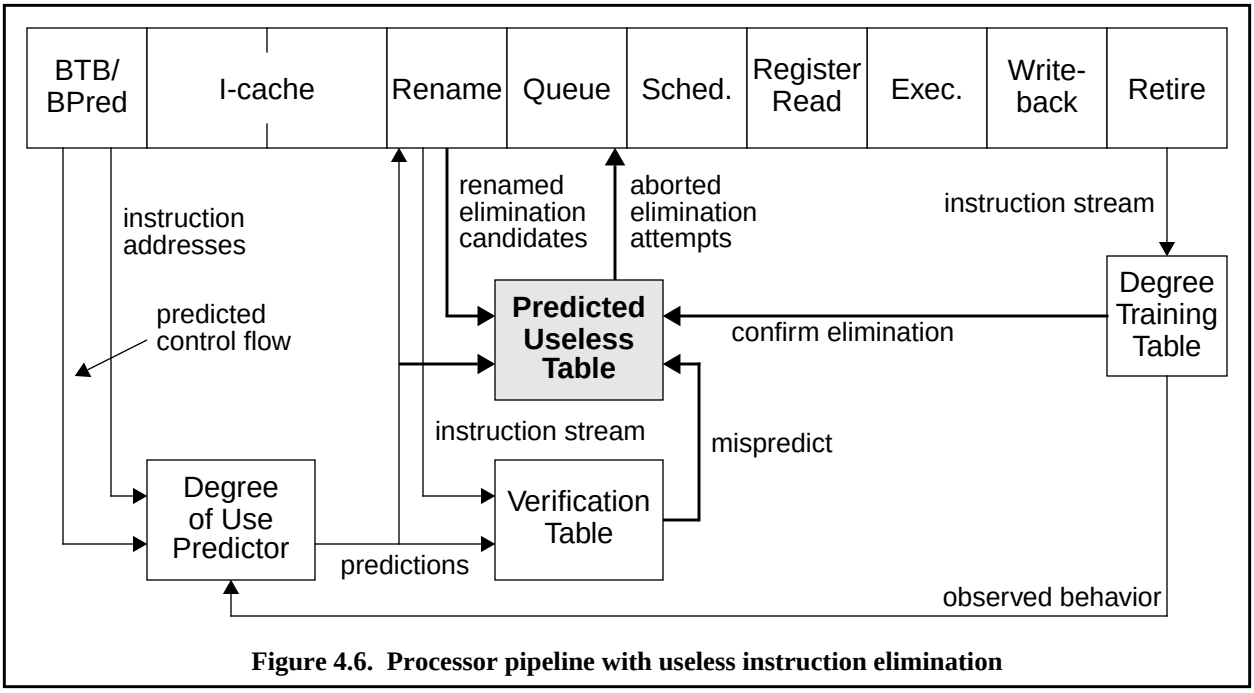


Figure 4.6. Processor pipeline with useless instruction elimination

addressed. First, however, some additional restrictions on the types of instructions eligible for elimination must be discussed.

4.2.1 Elimination candidates

To be a candidate for elimination, an instruction must be one that (1) generates a degree of use zero register value and (2) has no other side effects. Degree of use is only meaningful for those instructions that generate register results; therefore, a useless instruction must be one that computes a value. This requirement excludes nops and prefetches from the set of potentially useless instructions. Although most stores and control instructions do not generate a result value, the remainder are subject to degree of use prediction just like any other value-generating instruction. However, because of their side effects (modifying memory or the program counter, respectively), such instructions may be said to generate a useful result even when the destination register is not used. For example, while store instructions that generate a zero-use result may be truly useless if the stored data is never again referenced (or if the store is silent [55]), the detection of such instances (particularly in a multiprocessor machine) is more complicated, and outside the scope of register degree of use prediction.

4.2.2 Normal operation of useless instruction elimination

Useless instruction elimination begins with the identification of an eligible candidate instruction by the degree of use predictor (via a degree of use prediction of zero). The candidate instruction's source registers are renamed, and then it enters the PUT, which stores all eliminated instructions awaiting verification. Conceptually, each PUT entry consists of a valid bit, a decoded and renamed instruction, and a pointer to the reorder buffer (ROB) entry that would otherwise have contained the instruction. Eliminated instructions receive no physical register and they do not proceed to either the instruction window or reorder buffer; instead, a pointer to their PUT entry is placed in the reorder buffer as a placeholder. A new field added to each VT entry also contains a pointer into the PUT (recall that the VT structure is the part of the degree of use predictor used to verify predictions; see Section 3.4.7).†

The destination architectural registers of all instructions entering the rename stage are checked by the VT as part of its normal operation. When an overwrite of a predicted-useless value occurs and that value has a valid PUT pointer, the instruction being renamed is called the *verifying instruction*. The PUT pointer is copied into a field in the verifying instruction's reorder buffer entry. Before a placeholder instruction can retire, it must match its PUT pointer to that of a younger instruction in the reorder buffer (the verifying instruction). This matching operation is gated by the ready-to-retire status of each intervening instruction. When the verifying instruction and all older instructions are ready to retire, the placeholder retires and the corresponding PUT entry is freed. At that point, the instruction has been successfully eliminated.

Note that useless instruction elimination cannot rely on the degree of use predictor to verify a zero-use prediction. The VT must reside in the rename stage of the pipeline for misprediction detection (described in Section 4.2.3). Therefore, any verification of a zero-use prediction by the VT would be tentative because of the potential for wrong-path execution. Even if the VT (or a duplicate of it) were present in the retirement stage, the verifying instruction would have to retire to validate the zero-use prediction. This requirement conflicts with the need to avoid the retirement of the predicted-useless instruction until the prediction has been verified. The reorder buffer

---

† There is no specific requirement to use the PUT entry number—any means of uniquely identifying an in-flight instruction will serve. The ROB entry number, which must be assigned to elimination candidates, is one alternative. The PUT entry number will be slightly more efficient since there will be fewer PUT entries than ROB entries.

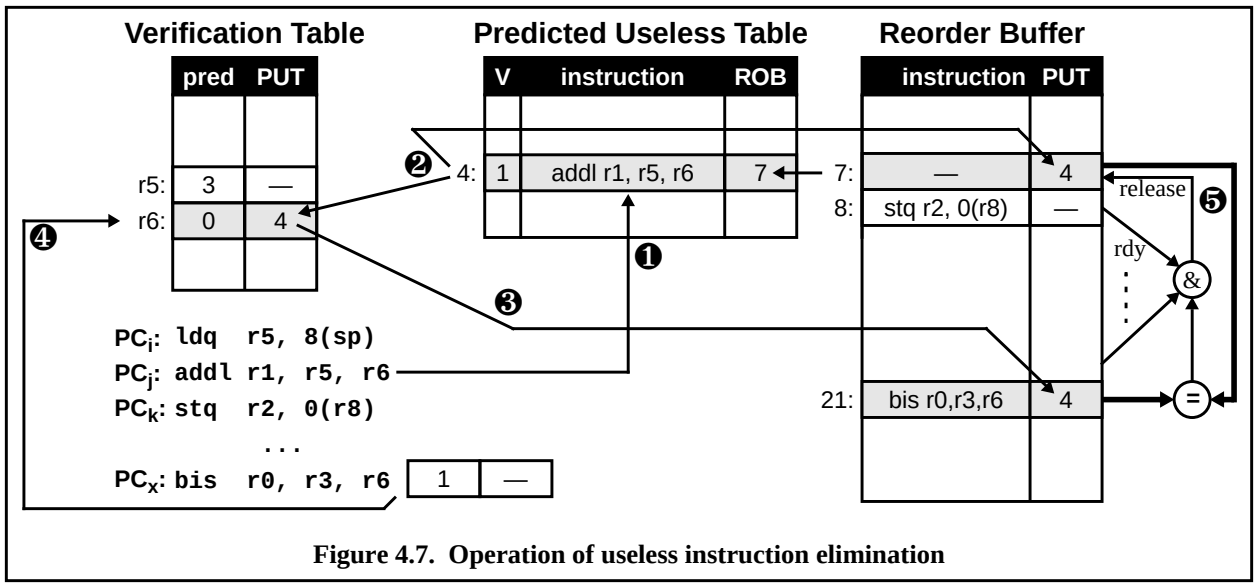


Figure 4.7. Operation of useless instruction elimination

matching operation solves the problem of prediction verification without the need to retire the predicted-useless instruction.

Figure 4.7 illustrates this entire process from prediction to verification. In this example, the value generated by the `addl` instruction at PC_j receives a degree of use prediction of zero. After renaming the instruction's source registers, it is placed into a free PUT entry (❶). A pointer to this PUT entry is placed into both the VT and the ROB (❷). The destination architectural register of the eliminated instruction determines the VT entry, while the ROB entry is the normal one for the candidate instruction. When the verifying instruction at PC_x is renamed, it is annotated with the PUT field from the VT entry (❸) prior to installing its own data in the VT (❹). Note that multiple instructions may receive this annotation due to wrong-path execution. When the verifying instruction and all intervening instructions are ready to retire, the speculation is verified and the PUT entry is freed (❺).

4.2.3 Misprediction detection and recovery

Misprediction recovery is straightforward. If the VT observes a use of a predicted-useless value, the incorrectly-eliminated instruction is fetched from the PUT (via the pointer in the VT), allocated a physical register, and inserted into the instruction window and the reorder buffer. Because a predicted-useless instruction is not allowed to retire before verification, the availability of its inputs is guaranteed. The physical registers containing the inputs must be freed by the retirement of an instruction later in program order than the predicted-useless instruction. While the physical

registers containing the inputs are guaranteed to be valid, it is quite likely that the *architectural* registers corresponding to the input values point to different physical registers (according to the rename map). The ephemeral nature of the rename map explains why the source registers of an elimination candidate are renamed normally.

An instruction naming a predicted-useless value as an input will not get a valid physical register until one has been assigned by the recovery process. Therefore, register renaming must be repeated when a misprediction is detected. The resultant pipeline bubble (potentially multiple cycles) conveniently provides a place to insert the mispredicted-useless instruction back into the pipeline. Ensuring that consumers of a mispredicted-useless value receive valid physical registers after renaming is why misprediction detection must occur at rename.

The number of instructions between a value's production and its first use can be used to estimate the misprediction detection latency. The median distance between the a value's definition and its first use is only four to six instructions (refer to Figure 2.4). Thus, most incorrect predictions can be detected very quickly. Note, however, that the misprediction penalty is *independent* of how long it takes to detect. Whether the misprediction is detected in ten cycles or a hundred, the cost is determined only by the time required to insert the mispredicted-useless instruction back into the pipeline. No instruction can have needed the value earlier than when the misprediction was detected.

Besides true mispredictions, exceptions can also be cause for misprediction recovery to be initiated. Synchronous exceptions such as a trap instruction require the retirement of all older instructions before the trap can be taken. These introduce a circular dependency: the predicted-useless instructions cannot be verified until the trap retires (and a subsequent verifying instruction is reached), and the trap cannot occur until the predicted-useless instructions retire. Therefore, upon encountering such an exception condition, the PUT must be completely emptied into the instruction window (i.e., a misprediction is signalled for each instruction in the PUT). The handling of asynchronous exceptions (e.g., external interrupts) can be handled in the same fashion. Alternatively, new additional eliminations can be halted and the exception handled after all pending eliminations have been verified or aborted (see Section 4.2.4). Still another alternative is to flush the entire pipeline, including the PUT, which would leave the processor in a consistent state.

4.2.4 Retirement backup

The verification process involves stalling the retirement of predicted-useless instructions until they can be verified. Therefore, the distance between consecutive writes to an architectural register (with no intervening reads) indicates the amount of speculative state that must be maintained in order to successfully eliminate a useless instruction. According to Figure 2.4, the median inter-definition instruction count is 21 instructions for zero-use values. Given a reorder buffer with more than 100 entries, one can expect that most useless instructions will not cause a problem. While the median def-overwrite distance is low, the tail of the distribution extends out to very long distances. Thus, there will be some small number of useless instructions that cannot be verified within the limits of any reasonable reorder buffer size.

Given the possibility of such unverifiable instructions, some additional mechanism must be in place to ensure forward progress. The simplest such mechanism simply detects the occurrence of this situation and initiates a recovery procedure as if a use of the unverifiable instruction had been encountered. Thus, the instruction will execute and retire normally, allowing the reorder buffer to drain and normal operation to proceed. These missed opportunities are called *aborted predictions*. Predictions may also need to be aborted to avoid deadlock conditions, which is discussed in Section 4.2.7.

Actually waiting until the reorder buffer is full before allowing the instruction to proceed carries an unreasonable performance penalty due to the resulting front end stall. A strategy that has been found to work well empirically initiates recovery (i.e., aborts the prediction) if the ROB capacity exceeds a threshold when the predicted-useless instruction becomes the oldest unretired instruction. The selection of this threshold involves a trade-off between the number of instructions eliminated and the performance cost of waiting for verification and is addressed in Section 4.3.1.

4.2.5 Loads

When an eliminated load is reinserted into the pipeline during recovery, it is delayed with respect to other loads and stores, which has implications for the memory consistency model. Out-of-order processors already solve this problem using mechanisms such as unified or separate load and store queues (LSQ) to maintain program order among memory operations to the same

address [39, 89]. To ensure that an eliminated load executes properly during misprediction recovery, the load must be placed into the load queue. Furthermore, the ordering and operation of a load queue frequently depends on the in-order allocation of entries to loads. Therefore, eliminated loads must, like all other loads, reserve entries to guard against the possibility of needing the services of the LSQ on a misprediction, reducing the benefit of eliminating useless loads.

One potential solution to this problem has been offered recently by Cain and Lipasti [19]. They eliminate the load queue altogether and re-execute selected loads at retirement to ensure that memory coherence and consistency are maintained. Such an underlying model is well-matched to the requirements of useless instruction elimination. Any mispredicted-useless load inserted back into the processor pipeline can be flagged to re-execute at retirement to ensure correctness without requiring the load to enter a load queue or check the store queue. Under certain circumstances, such as when the load is the oldest instruction (e.g., due to an aborted prediction) and no external memory references have occurred, the check may be safely skipped. Even when the underlying implementation uses a load queue, re-execution can be applied specifically in the case of mispredicted-useless loads. Such a scheme extends the resource reduction benefit of useless instruction elimination to the load queue, but may reduce performance if load replays are frequent among the reinserted loads (a load replay being the action taken to recover from a load that got the incorrect data—often a squash and refetch of all instructions younger than the load). While this technique may be beneficial, the simple solution of requiring loads to reserve an LSQ entry is assumed in this chapter.

4.2.6 Instructions with side effects

Another subtle problem with the elimination of loads arises when loads with side effects are considered. Device drivers, for example, may depend on loads to certain memory-mapped hardware addresses. In this case, the action of the load on the hardware state (rather than the use of the loaded value) may be the purpose of the load. Loads causing page faults or loads to intentionally out-of-bounds addresses raise exceptions, which are architecturally-visible side effects of their execution. Instructions other than loads may also raise exceptions. Arithmetic instructions use exceptions to signal divide-by-zero, overflow, and underflow conditions. Instructions with potential side effects cannot be unconditionally eliminated without breaking architectural compatibility.

There are two broad solutions to the issue of eliminating instructions with side effects. First, the potential for the occurrence of an exception or other side effect can be ruled out before an instruction is eliminated. The usefulness of this method depends on the check for side effects requiring less effort than the complete execution; otherwise, the subsequent elimination would not offer any benefit. The other possibility is to define instructions that cannot raise exceptions or execution modes in which the exceptions are ignored. This solution allows for the elimination of instructions without the burden of verifying that an exception (or other side effect) will not occur, but requires support in the architecture.

Checking for side effects before useless instruction elimination is probably best-suited to load instructions. Note that any load that could potentially be satisfied from the cache cannot have a side effect. By accessesing the TLB for each eliminated load, it is possible to ensure that the load address belongs to a cacheable page. A TLB miss would result in a page fault and the scheduling of the load for execution. This solution unfortunately requires address computation to be performed on eliminated loads. However, it is guaranteed to be safe, it does not require any architectural support, and it still eliminates the need to perform the cache access.

In the case of arithmetic operations, checking that an instruction is exception-free can be of comparable complexity to the execution itself. Overflow detection for an integer addition, for example, requires the computation of the most significant carry bits, which is almost as difficult as the complete addition. Checking for division by zero is simpler, but still requires that one of the useless instruction's operands be read.

Avoiding the expense of verifying that an elimination candidate is free from potential side effects is possible with some architectural support. Different flavors of instructions can be provided that communicate the importance (or irrelevance) of an instruction's exception behavior to the hardware (i.e., the elimination mechanism). Instructions for which the exception behavior is unimportant may be safely eliminated without any further checking. The Alpha ISA, for example, already provides arithmetic instructions that differ in their ability to signal overflow and other exception conditions.

Where the instruction encodings cannot be changed or backward compatibility must be maintained, it is possible to define a new execution mode in which side effects are not guaranteed to occur for eliminated instructions. Programs making use of this mode communicate to the hard-

ware that their operation does not depend on the exception behavior (or other side effects), allowing for the elimination of arbitrary instructions. Since the SPEC benchmarks do not depend on any exception behaviors or other instruction side effects, operation in such a mode is assumed in this chapter to demonstrate the overall potential of the technique.

4.2.7 Deadlock avoidance

The conservation of resources, execution and otherwise, and the accompanying reduction in contention are the benefits of useless instruction elimination. When a predicted-useless instruction needs to re-enter the normal execution pipeline (e.g., due to a misprediction or aborted prediction), however, it will require the resources not initially allocated it. If freeing one or more of these resources depends upon the execution of the lately-inserted instruction, deadlock can result.

Deadlock can also occur when a predicted-useless instruction awaiting verification is blocking retirement while the front end is stalled due to resource exhaustion (e.g., lack of physical registers, load-store queue entries, reservation stations, etc.). These types of deadlocks are easily detected; by aborting the elimination of the instruction blocking retirement, the situation frequently reverts to the other type of deadlock wherein the aborted prediction requires additional resource allocation. Therefore, the existence of an *automatic abort mechanism* is assumed, which detects and aborts an unverified elimination blocking retirement (i.e., at the head of the reorder buffer) when any resource is exhausted.

The only resources that can lead to deadlock are those that are held by a younger instruction through its retirement—execution resources, for example, cannot cause a problem. In the implementation of useless instruction elimination described in this chapter, only physical registers meet this condition (recall that load-store queue entries are assigned to eliminated loads; see Section 4.2.5). Furthermore, the deadlock situation requires an unverified useless instruction at the head of the reorder buffer (i.e., as the oldest in-flight instruction); otherwise, retirement will not be blocked and physical registers will eventually be freed.

Since the deadlock case will be rare, one solution simply detects the situation and forces a squash as if the most recent branch had been mispredicted (additional recent branches or even the entire pipeline can be squashed if no physical registers are freed by the first attempt). If a dead-

lock occurs, the particular sequence of instructions will always result in the same situation, so an additional mechanism would be required to prevent recurrences.

An alternative solution to the deadlock problem simply provides enough physical registers to guarantee that each predicted-useless instruction can always get a physical register. It is not required that a physical register actually be allocated (implying modification of the freelist, etc.), just that at least one physical register per in-flight elimination candidate is kept free. While allocation and freelist management overhead and register file write bandwidth are reduced, no savings on physical register file occupancy is observed because the useless values still, in effect, occupy an entry. Because of its simplicity, however, this method is assumed in the evaluation of Section 4.3.

A slight modification (not evaluated) reduces the number of "reserved" physical registers to one for the oldest predicted-useless instruction. Any time an abort or misprediction of any *other* predicted-useless instruction would require this last reserved register, the oldest pending elimination is aborted instead (consuming the final physical register). As that instruction completes, it will likely lead to the retirement of multiple instructions, freeing many additional physical registers; at a minimum, however, one physical register would be freed, allowing the process to continue. This optimization is subsumed by the aforementioned automatic abort mechanism if the reserved physical register is excluded when determining resource exhaustion.

4.3 Results

This section presents an evaluation of useless instruction elimination via full timing simulation. Parameters of the simulated processor appear in Table 4.3. Additional details on the simulator may be found in Section A.4 of the appendix. Note the relatively large reorder buffer and physical register file sizes, which support deep speculation (execution proceeding far ahead of retirement). While most of the benefit of useless instruction elimination can be obtained with more reasonably-sized structures, these sizes allow for the study of a larger range of behaviors.

Useless instruction elimination can only improve performance when there is resource contention to begin with. For this reason, three different sets of execution resources were evaluated with the other microarchitectural parameters fixed. Table 4.4 summarizes the functional unit resources and issue port bindings for each case. Ready instructions are issued oldest-first to the

Table 4.3: Simulated Processor Parameters

Pipeline	4-wide superscalar; 3-stage fetch (next address + I-cache access + fetch queue), 1-stage each decode, rename, dispatch (write into window), issue, register file read, register file write, and commit. Variable execution latency, delay between issue and execute, and delay between register file write and commit. 10-cycle minimum fetch redirection on branch mis-speculation.
Front end	Up to four non-nop instructions per cycle; taken branch (including unconditional) or cache line boundary terminates fetch. 16-entry instruction queue between L1 I-cache and decode.
Issue	64-entry scheduling window, oldest ready first. 256-entry reorder buffer, 256 physical registers. Issue port bindings as described in Table 4.4.
Execute	4-cycle latency integer multiply, 2-cycle store latency (to detection of ordering violations and ability to supply subsequent loads), 3-cycle load to use latency on L1 hit, 2-cycle branch, 4-cycle FP multiply, 16-cycle FP divide, 33-cycle FP sqrt; all other integer operations 1 cycle, FP operations 2 cycles. Execution resources as in Table 4.4.
Memory	64KB, 2-way set-associative L1 inst. and data caches with 64-byte blocks. 2MB, 4-way set-associative unified L2 cache with 128-byte blocks, 8-cycle latency. 100-cycle memory. 64-entry load queue and 64-entry store queue.
Degree of Use Predictor	8K-entry, 8-way set-associative, 13.4KB predictor described on page 80.

Table 4.4: Functional Unit and Issue Port Configurations

Issue port	Rich	Medium	Scarce
1	Simple integer ALU op. (not ld/st/branch/mult)	Simple integer ALU op.	Any integer ALU op.
2	Any integer ALU op.	Any integer ALU op.	Any FP operation
3	Simple integer ALU op, load, or store	Any FP operation	Load or store
4	Load or store	Load	Integer branch
5	Integer branch	Integer branch or store	
6	FP except mult/div/sqrt		
7	Any FP operation		

first (lowest-numbered) issue port with an appropriate ALU. The maximum issue bandwidth equals the number of issue ports (i.e., 7, 5, and 4 instructions per cycle for the rich, medium, and scarce configurations, respectively). The rich configuration was obtained by considering combinations of operations likely to be found in a group of four instructions without regard to the issue width. The scarce configuration matches that of the Transmeta Crusoe [51], which is quite resource-constrained for a four-issue machine, while the medium configuration represents an intermediate design point.

4.3.1 Parameter sensitivity analysis

The most important parameter of the useless instruction elimination mechanism is the ROB fill threshold for aborting predictions. Figure 4.8(a) shows how this parameter affects the percentage of useless instructions eliminated for each of the three resource configurations. The PUT size is fixed at 64 entries for this experiment. Note that a threshold of zero still results in 50–65% of all possible eliminations. At a threshold of zero, verification must be possible immediately when the predicted-useless instruction reaches the head of the ROB or the elimination will be aborted. Increasing the ROB threshold results in more eliminated instructions up to a threshold of about 192 (out of a ROB size of 256), where approximately 80% of useless instructions are eliminated.

While the portion of useless instructions eliminated rises monotonically with the threshold, the performance reacts quite differently, as shown in Figure 4.8(b). The threshold offering the highest performance for each configuration is indicated with an arrow. First, note the relative positions of the curves, which indicate how the benefit (or cost) of useless instruction elimination depends on underlying contention. The scarce configuration benefits over the range of thresholds; in contrast, the rich model exhibits a performance *loss* everywhere. Where contention is not a performance limiter, there is no gain to offset losses due to misidentified useless instructions or the backup of retirement; therefore, the minimum threshold (leading to the fewest elimination attempts) offers the minimum performance loss. This loss was primarily a result of two poorly-performing benchmarks—eight of the other ten showed a very small speedup at a threshold of zero. These problem benchmarks will be discussed further in Section 4.3.4.

For the other two configurations, modest performance improvements are achievable in a manner that depends on the threshold. At very low thresholds, the primary cost of useless instruction

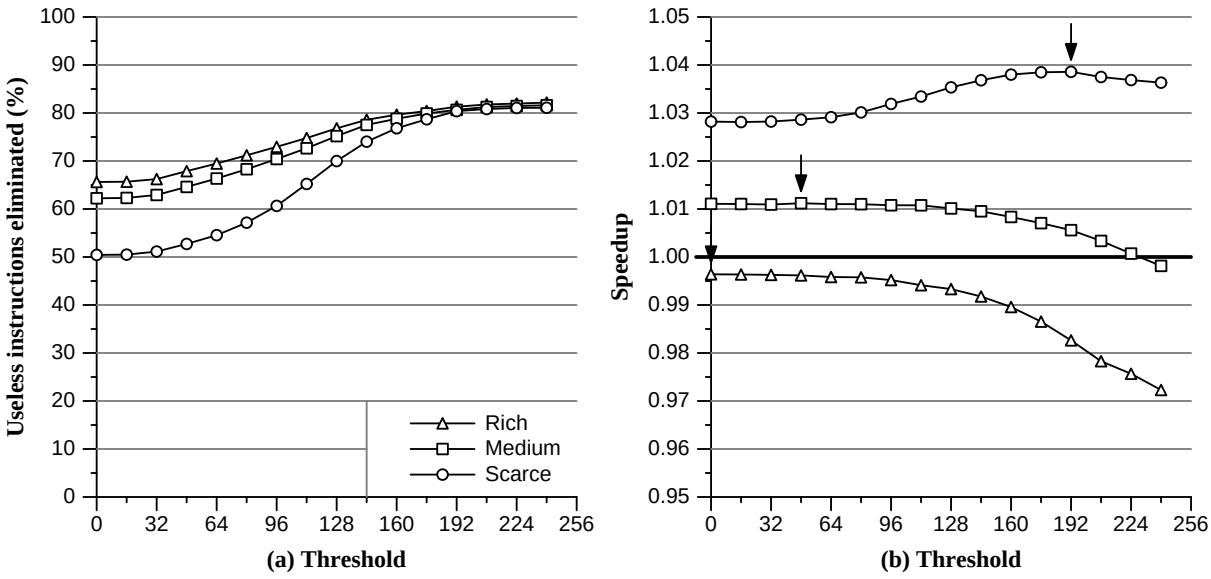


Figure 4.8. ROB fill threshold sensitivity

elimination is the delay of useful instructions mistakenly identified as useless. To first order, this cost is independent of the threshold. As the threshold increases, the probability of resource exhaustion increases along with the attendant front end stalls. At some threshold value, the increasing costs of the retirement backup and front end stalls exceed the benefit of the additional eliminations. Resource contention shifts the optimal threshold to higher values as the marginal benefit of each elimination is higher. The figure shows that the overall peak performance occurs at a threshold of 48 and 192 for the medium and scarce configurations, respectively. Note that the best threshold for a given benchmark may deviate significantly from the overall optimum, making the elimination threshold an ideal candidate for dynamic tuning [25].

Figure 4.9 is similar to Figure 4.8, but shows the effect of the PUT size at a fixed threshold of 128. The behavior here is much more straightforward. The size of the PUT determines the maximum number of pending eliminations in flight. Therefore, limiting the size of the PUT will reduce the fraction of useless instructions eliminated. Unlike the similar effect when the ROB threshold is reduced, however, elimination opportunities are lost indiscriminately—the missed opportunity may be an easy to verify useless instruction or a difficult one. Therefore, in those configurations where useless instruction elimination is beneficial at all, increasing the size of the PUT always yields more benefit. The reverse is true for baseline configurations where useless instruction elimination is detrimental.

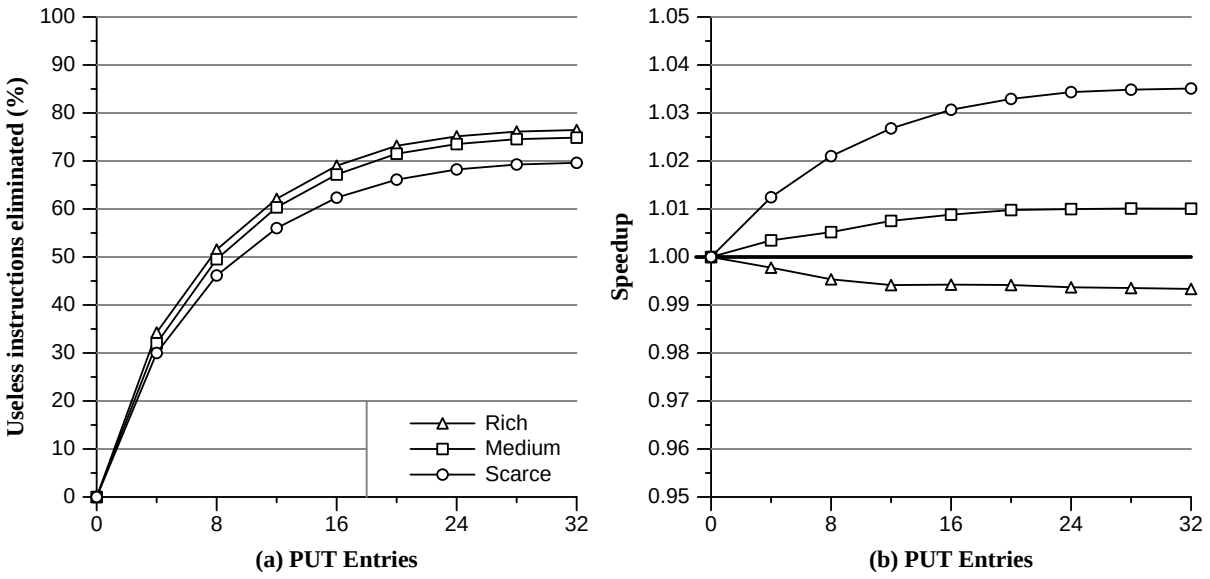


Figure 4.9. PUT size sensitivity

The choice of PUT size should therefore be dictated by hardware cost considerations. The maximum number of instructions in flight (i.e., the ROB size) and the incidence of useless instructions bound the maximum possible PUT occupancy; increasing the PUT size beyond this point has no effect. For the pipeline studied here, this occurs at a capacity of around 32 entries. The remainder of the experiments in this chapter assume a PUT size of 64 entries (which gives nearly identical results to a 32-entry PUT) and a ROB threshold of 192.

4.3.2 Resource utilization

Table 4.5 shows the percentage of useless instructions eliminated and the resulting percentage reductions in resource utilization. The medium execution resource configuration was used in this experiment. Similar results are seen with the other execution configurations although the savings increase slightly as the available execution resources are increased (also visible in Figure 4.8(a)). About 80% of the useless instructions accounting for over 5% of all dynamic instructions are eliminated successfully. Resource utilization is decreased by approximately the same magnitude. Four of the twelve benchmarks realize reductions of over 10% in either register writes or cache accesses.

Differences in the relative reductions of executions, register reads and writes, and cache accesses depend on the specific instruction mix of the benchmark. Benchmarks in which useless

Table 4.5: Utilization Impact of Useless Instruction Elimination

Benchmark	Useless inst's eliminated	Executed instructions	Register file reads	Register file writes	Data cache reads
bzip2	87.35	–1.38	–0.73	–1.64	–2.30
crafty	84.71	–7.60	–6.56	–8.79	–11.28
eon	68.29	–5.57	–4.32	–7.38	–6.05
gap	91.47	–8.76	–5.46	–11.09	–6.40
gcc	78.90	–6.60	–4.53	–9.09	–7.26
gzip	89.80	–7.16	–5.40	–8.88	–10.04
mcf	84.08	–6.49	–4.89	–8.22	–3.99
parser	74.87	–3.60	–2.54	–5.02	–4.13
perl	83.27	–6.05	–4.23	–7.86	–5.97
twolf	75.17	–1.62	–1.14	–2.02	–2.92
vortex	84.76	–9.25	–5.79	–12.86	–8.52
vpr	65.16	–3.30	–2.54	–4.05	–5.44
Average	**80.65**	**–5.61**	**–4.01**	**–7.24**	**–6.19**

instructions have proportionally more loads than the overall instruction mix (see Table 4.1) exhibit an amplified reduction in the number of cache accesses relative to the number of executions eliminated (e.g., `crafty`). The mix of zero, one, and two input instructions (see Table 4.2) affects the relative decrease in register reads. Because useless instructions have fewer register inputs than the average for all instructions, the reduction in register reads is always less than the reduction in executions. Register writes, on the other hand, are always decreased relatively more than executions since not all instructions generate values (e.g., branches and stores), but all eliminated instructions do.

Figure 4.10 shows the disposition of the retired useless instructions. While nearly 90% of useless instructions are identified by the predictor (i.e., the predictor's coverage on degree of use zero values), only 81% of them are successfully eliminated. Three events prevent a predicted-useless instruction from being eliminated: (1) the ROB fill threshold was exceeded before the prediction could be verified (aborted predictions), (2) a prediction was aborted due to resource exhaustion (physical registers or LSQ entries), or (3) an apparent use of the value produced by the instruction was observed on a wrong path. Of these causes, the first two are the most important,

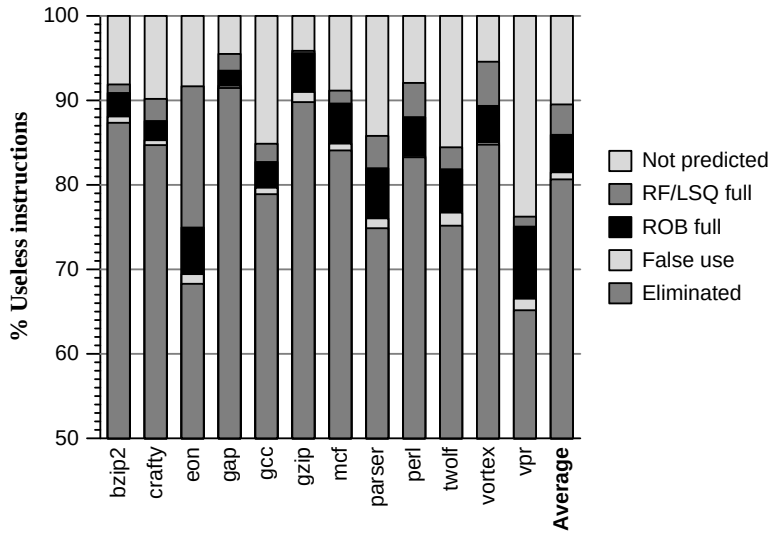


Figure 4.10. Retired useless instructions

accounting for 90% of the non-eliminated correct predictions. Increasing the ROB threshold reduces the number of aborted predictions, but increases the incidence of resource exhaustion. The sum of the two effects results in a net decrease of aborted predictions, however, as reflected in the increased number of successful eliminations. The non-prediction rate and the rate of eliminations aborted due to false uses are approximately constant with threshold.

4.3.3 Resource occupancy

The need to stall retirement pending the verification of eliminations increases the occupancy structures required to track in-flight instructions, which include the ROB, the load and store queues, and the register file. Figure 4.11 shows how the average occupancy of these structures depends on the threshold. The data are normalized to that seen without useless instruction elimination.

Employing useless instruction elimination increases the occupancy of the ROB and the load and store queues by a minimum of 40%. Even when unverified eliminations are aborted immediately upon reaching the head of the ROB (i.e., becoming the oldest instruction), retirement is stalled until the aborted instruction can be scheduled and executed. Thus, occupancy is increased even for a the ROB threshold of zero. The relative increases depend primarily on the pipeline depth: deeper pipelines increase the abort delay, allowing more instructions to claim resources during the retirement stall. In this pipeline configuration, occupancy increases of 80% were

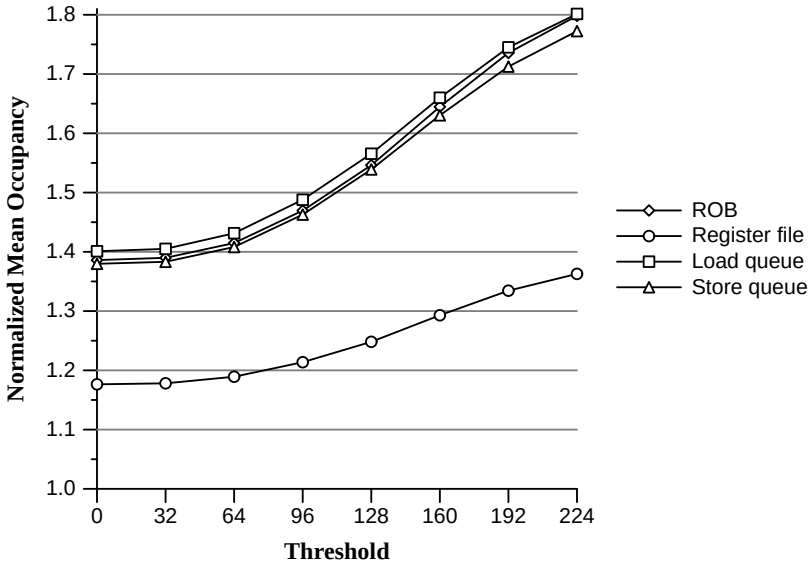


Figure 4.11. Resource occupancy

observed at higher values of the ROB abort threshold. The smaller increases in register file occupancy result from the fact that a fixed number of physical registers (62) are always allocated to contain the architected register state even when there are no in-flight instructions.

4.3.4 Performance

Figure 4.12 delves further into the performance effects of useless instruction elimination on a per-benchmark basis. For each benchmark, a stacked bar represents the speedup for the Rich, Medium, and Scarce resource configurations over a machine without useless instruction elimination. The dark gray bar indicates the actual performance with the real degree of use predictor described in Table 4.3. The lighter gray bar corresponds to the performance of useless instruction elimination with a perfect degree of use predictor, but where each prediction must still be verified normally (i.e., within the constraints of the ROB threshold and with the associated retirement backup). Finally, the black bar indicates the performance with a perfect predictor where elimination candidates retire immediately. Where the dark gray bar appears completely absent (e.g., in the R configuration of `gzip`), the performance of the real predictor is actually *higher* than the performance with a perfect predictor, but the difference is so small that the bar is not discernible. The analogous case where the light gray bar appears missing (e.g., in the R configuration of `gap`) indicates that the performance of a perfect predictor barely edges out the performance of the real

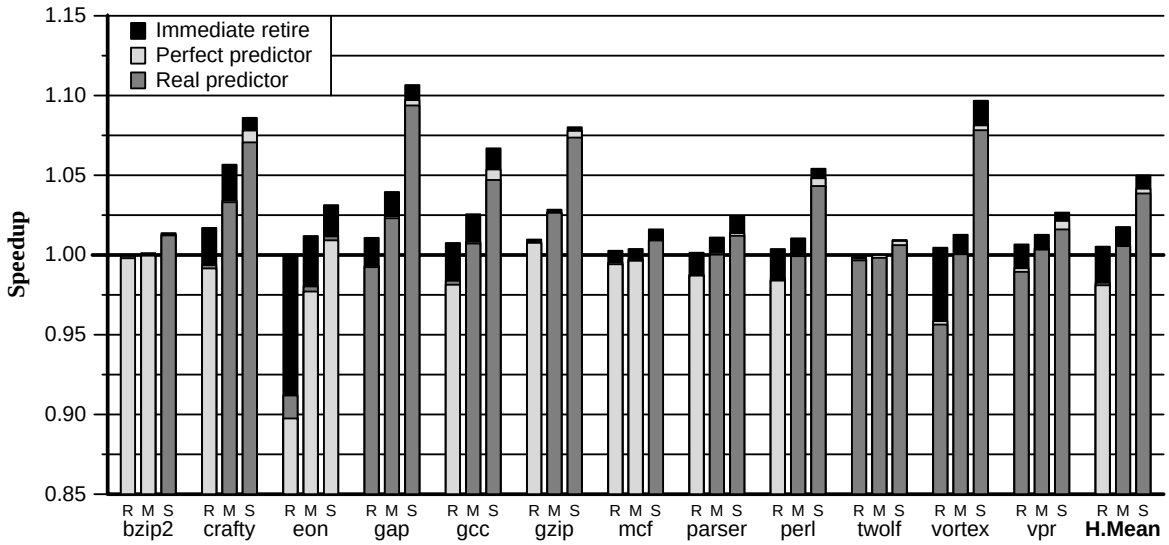


Figure 4.12. Performance

predictor. In all cases, the lowest performing configuration is indicated by the bar extending to the bottom of the figure.

Broken down in this manner, the performance data clearly illustrate the interplay between the two possible performance penalties of eliminating useless instructions. In nearly every case, the performance with a real predictor is very close to that of the performance with a perfect predictor, indicating that the cost of degree of use mispredictions is generally negligible, especially since the perfect predictor also detects additional useless instructions. The main cost of useless instruction elimination is clearly the need to verify the predictions and the associated consequences. The verification cost is almost entirely responsible for the performance loss experienced by the resource-rich model. The high cost of verification also explains why a perfect predictor can perform worse than a real predictor—the costs of verifying the additional useless instructions identified exceed the miniscule savings from the extra eliminations.

The two benchmarks contributing most to the slowdown of the rich configuration are `eon` and `vortex`. Examining the behavior of these two benchmarks further reveals that predictions are aborted more frequently due to a full load or store queue than due to the ROB threshold. This behavior is confirmed in Figure 4.10 for the medium configuration. As discussed in Section 4.2.7, resource exhaustion requires an abort to avoid deadlock. The entire front end is stalled until the aborted instruction can execute and retire and free up the necessary resource. Lower thresholds help these benchmarks immensely because it reduces the probability that a

resource will run out prior to the threshold being exceeded. The mechanism of useless instruction elimination could be improved by setting abort thresholds analogous to the ROB fill threshold on the register file and load and store queue occupancies.

4.4 Related Work

There is an extensive body of work on the compile-time detection and elimination of dead and partially-dead instructions. Partial dead-code elimination [53] is a compiler algorithm that transforms code to reduce or eliminate instances of instructions that produce dead values. In essence, the algorithm detects code that generates values used on only some subsequent control flow paths and attempts to move that code down into those paths. Improvements to the algorithm have been offered to widen its applicability while ensuring that the number of instructions on a given path is never increased [9]. As noted in Section 4.1.3, however, the applicability of static schemes is limited as many partially-dead instructions arise from beneficial optimizations that are, in effect, the reverse of partial dead-code elimination.

Martin et al. proposed a cooperative software/hardware scheme to track registers containing dead values [60]. Their scheme involves annotating the executable with information from the compiler about the last use of register values (i.e., noting that specific registers are dead). This information is subsequently used by the hardware to enable early physical register reclamation and elimination of needless saves and restores across procedure calls and context switches. Useless instruction elimination cannot eliminate the saves of dead values (since stores have side effects), but it is able to eliminate useless instructions within procedures, which are not handled by this scheme.

Yoaz et al. [91] noted the occurrence of useless instructions, focusing on the subclass of them called silent stores [55]. They mentioned the possibility of squashing or de-prioritizing dead instructions, but did not present any specific schemes to identify or handle them. Rotenberg also observed the occurrence of sequences of instructions with no externally visible effects [71]. A much broader category than useless instructions, his *ineffectual instructions* include dead instructions, silent instructions (stores and otherwise), correctly predicted branches, and instructions transitively connected only to other ineffectual instructions. He proposed skipping these instructions to allow a speculative thread to get ahead of a slower verification thread.

4.5 Summary

Useless instructions generate result values that are not subsequently used during a program's execution. These instructions can account for as many as 15% of the dynamic instructions in an optimized program. The execution of useless instructions wastes processor resources, increasing utilization with no effect on the final computation. Resources used include physical registers, load queue entries, execution units, issue bandwidth, register file read and write bandwidth, and data cache bandwidth. A direct consequence of this increased resource utilization is higher-than-necessary power consumption. Under resource contention, performance suffers as well.

The compiler is directly or indirectly involved in the existence of all useless instructions. A small portion of useless instructions are the result of dead instructions undetected by the compiler; these instances could be eliminated given more powerful compiler analysis. Most useless instructions, however, arise from partially-dead static instructions introduced by the compiler as a side effect of optimizations that involve code motion. As these optimizations generally improve performance, useless instructions are likely to remain in optimized programs regardless of advances in compiler technology.

Useless instruction elimination is a scheme to avoid the execution of these instructions, reducing the utilization of several key resources. The mechanism of useless instruction elimination is straightforward. Candidates for elimination are identified by degree of use prediction prior to consuming most resources. These instructions are shunted into a special structure to await confirmation of their status by the execution of instructions that overwrite their results. When a candidate instruction's value is overwritten (prior to being used) and that overwrite is known to be on the correct execution path, the predicted-useless instruction may be discarded. Recovery from a misprediction simply requires executing the incorrectly-eliminated instruction.

Up to about 80% of the useless instructions in the benchmarks—accounting for 5.6% of all dynamic instructions—can be eliminated using this technique. Attendant reductions in register reads, register writes, and L1 cache accesses of 4%, 7%, and 6%, respectively, were also observed. One in four of the benchmarks realized a reduction of more than 10% in either register file writes or cache accesses.

The performance impact of useless instruction elimination is heavily dependent on the contention for execution resources. An average speedup of 3.9% was obtained (with four benchmarks

exceeding 7% speedup) on an implementation suffering from resource contention. On an execution-resource-rich implementation, however, an average performance loss of about 0.4% was observed.

Performance losses are primarily a result of front end stalls resulting from resource exhaustion. Useless instruction elimination is a deeply-speculative operation: execution can proceed far ahead of retirement while an unverified elimination is pending. The large number of in-flight instructions this implies require physical registers and load and store queue entries. If one of these resources is consumed before the oldest useless instruction is verified, the entire processor stalls while the elimination is aborted and retired, allowing resources to be reclaimed.

There are several ways in which the mechanism of useless instruction could be improved. One optimization already alluded to in Section 4.3.4 is the use of thresholds on other consumable resources to avoid the aforementioned resource stalls. Addition of state to the degree of use predictor could identify difficult-to-verify instructions so elimination would not be attempted. The use of a checkpointing mechanism [3] would allow for deep speculation without a retirement backup. Useless instructions could be discarded immediately, and the PUT and retirement verification logic would be completely eliminated. Such an implementation would make eliminations very cheap at the cost of much a more expensive recovery operation on mispredictions, favoring predictor accuracy over coverage. Each of these improvements attempts to address the costs associated with useless instruction elimination. One possible change to improve the benefit would be to avoid the assignment of load queue entries and physical registers to elimination candidates as suggested in Section 4.2.5 and Section 4.2.7. The pursuit of these ideas is left to future work.

Chapter 5

Use-Based Register Caching

This chapter presents the application of degree of use prediction to register cache management. A register cache is a small structure that maintains a selected subset of the values generated by execution. By virtue of its selective nature, it can be made small, allowing lower access latency than a full-sized register file. Together with a standard bypass network, the register cache comprises an alternative inter-instruction communication mechanism that is more efficient than a monolithic register file.

Degree of use prediction provides the information to determine which values should occupy the limited storage available in the cache. By comparing the degree of use of each value with the number of uses that have actually occurred, the occurrence of future uses can be predicted. This chapter describes insertion and replacement policies that use this knowledge to keep the most pertinent values in the register cache—namely, those with outstanding consumers.

5.1 Introduction

The register file, by definition, is the predominant value communication mechanism in a sequential, register-based architecture. However, it is becoming increasingly difficult to support a large, low-latency, monolithic physical register files in high-performance superscalar implementations. These processors are likely to have deep pipelines [37, 41, 79] and be multiple issue, resulting in a large number of instructions in flight, most of which require a register for their result. Simulta-

neously, enabled by technology improvements and a deeper pipeline, clock frequency is increasing, decreasing the amount of state that can be addressed in a fixed number of cycles [1]. The result is increased register file read and write latencies. The read latency is particularly problematic since it appears in both the branch misprediction and load-hit speculation loops [11]. Furthermore, to allow unrestricted issue of dependent operations, the total number of stages in the bypass network must increase with the register file latency. Bypass networks are dominated by long wires and wide multiplexors, which do not scale well to high frequencies. A limited bypass network [2] adds to the performance impact of a multi-cycle register file access.

The key to solving the register file problem is to recognize that the register file is performing two other functions besides supplying instruction input values. First, the register file participates indirectly in maintaining inter-instruction dependencies by supplying a namespace of physical register tags. Architectural registers are renamed to physical registers early in the pipeline, requiring the physical registers to be allocated at that time. As a side effect, many entries in the register file are *empty*, being allocated to instructions that have not yet produced a result. Second, in order to support recovery from mis-speculation, the physical register file maintains instruction result values long after the final consumers have obtained that result. These *dead* values, together with the empty registers, vastly inflate the capacity of the register file beyond what is required for its most important role—value communication.

Figure 5.1 illustrates the combined impact of empty registers and dead values (the simulated machine is described in Section 5.4.1; basically it is an aggressive, deeply-pipelined, eight-wide superscalar machine). From bottom to top, the bar for each benchmark indicates the number of registers associated with live values, no value (i.e., empty registers), and dead values; the total height of the bar equals the average total number of allocated registers.† Clearly, the register file could be made much smaller—and faster—if its contents were limited to the live values.

However, one can do even better by recognizing that most processors already employ an alternate communication mechanism—the bypass network. Values that communicate to a small number of consumers shortly after being generated can be handled completely within the bypass network, avoiding the need to store even some live values. To illustrate the potential savings,

† Register file write latency is ignored here when classifying registers. A register is considered live as soon as the instruction generating its contents executes.

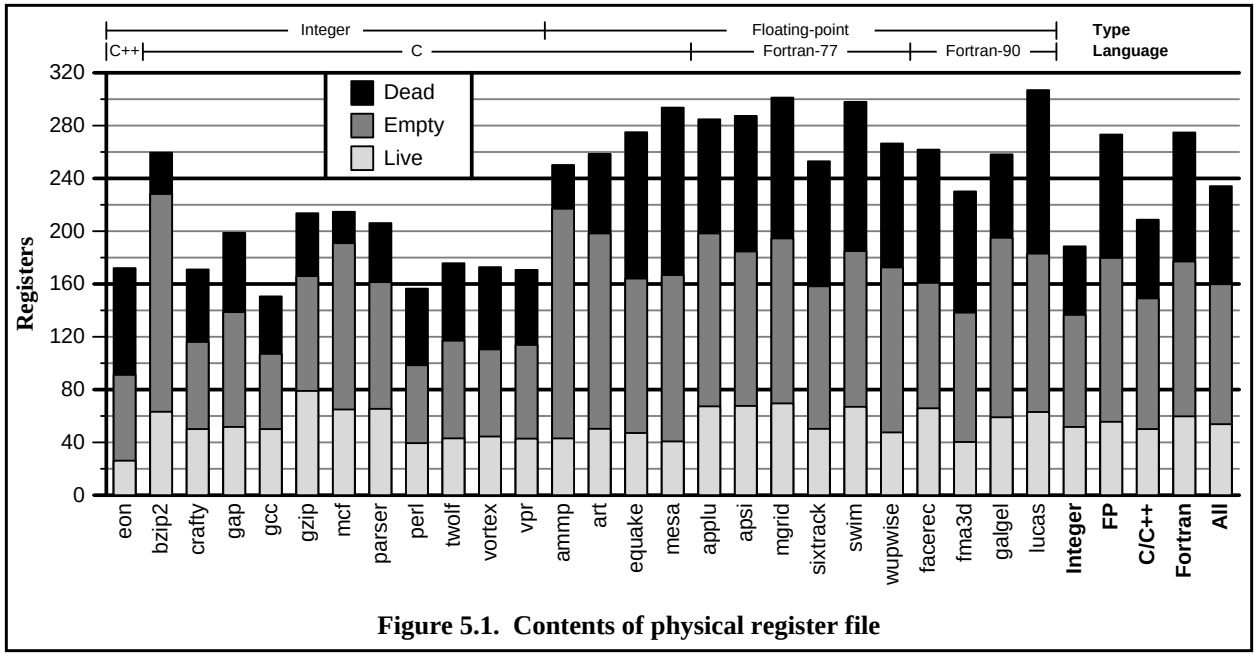


Figure 5.1. Contents of physical register file

Figure 5.2 indicates the fraction of all register file reads satisfied by each stage of the bypass network. The simulated processor is identical to that of Figure 5.1, which has a three-cycle register file (read and write) that is fully-bypassed (requiring six total bypass stages). The components of each bar indicate the fraction of input values obtained from the bypass network from the first stage (i.e., bypasses from instructions executing in the immediately preceding cycle) at the bottom to sixth stage on the top. The remainder of the values come from the register file.

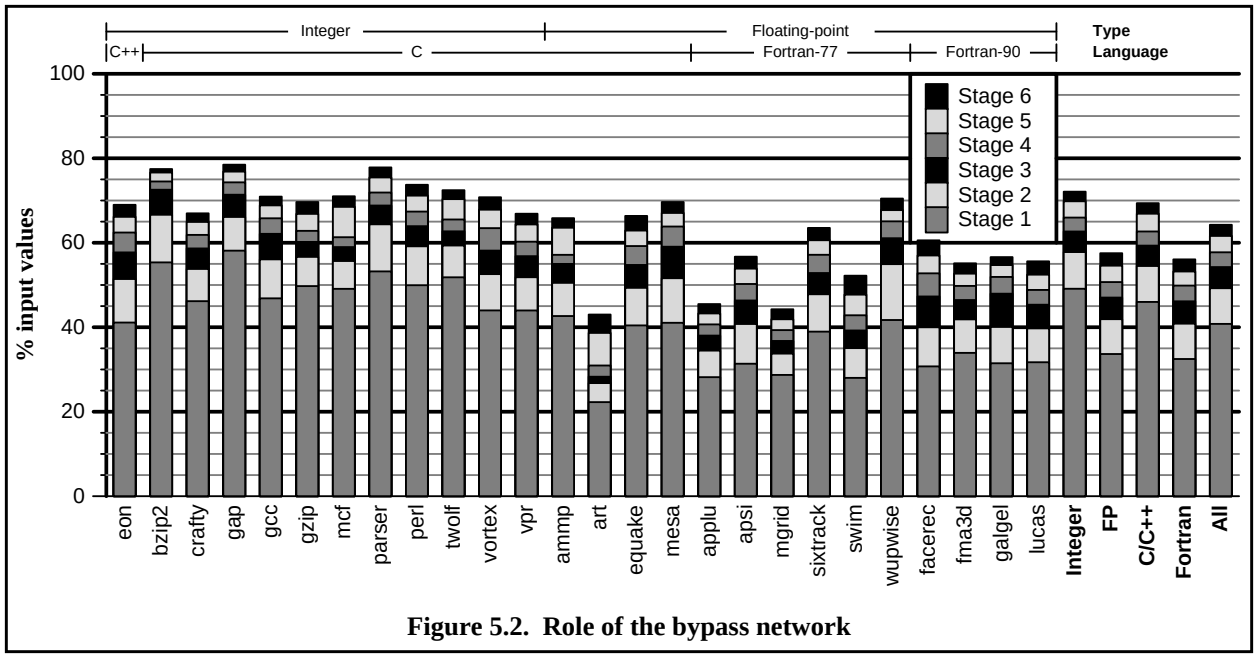


Figure 5.2. Role of the bypass network

In all but three benchmarks, over half of input values are bypassed. The first bypass stage is the most important, accounting for the majority of bypassed values in all cases. Later stages bypass successively fewer values in general. Floating-point benchmarks bypass fewer values than integer ones because floating-point values tend to be longer-lived (see Figure 2.4), supplying many consumers beyond the reach of the bypass network. The low average degree of use combined with the large number of reads occurring before the register file is able to supply values suggest that many of the live values counted in Figure 5.1 are effectively dead within the register file.

These observations motivate the replacement of the register file by a small *register cache* [92]. The register cache assumes the responsibility for handling register read requests by the execution core. While the register cache still needs a bypass network, the network can be much smaller than needed for a multi-cycle register file because of the register cache's lower latency. Also, the register cache can work cooperatively with the bypass network, accounting for bypassed values in managing the cache contents. The reduced latency to obtain most values shortens the execution pipeline providing higher performance.

Degree of use prediction is fundamental to the operation of this register cache, improving its performance over prior register cache proposals [11, 24, 92]. A value's degree of use is exactly the number of reads that are expected for that value. This information is used to manage both the insertion and replacement of values within the cache. The insertion policy identifies values that have bypassed to all of their expected consumers, and avoids placing them into the cache. The remaining-use count is kept for each cached register value and updated as uses are satisfied by the cache. When a replacement is necessary, the cached value with the fewest remaining uses (ideally zero) is selected as a victim, minimizing the potential for misses resulting from the replacement.

An overview of the operation of these policies appears in Figure 5.3. Each box represents a register value; the corresponding physical register tag is indicated by p and the number of remaining uses follows in parentheses. The value itself is immaterial and is not shown. On the left hand side are values generated by the execution core, which each bypass to a consumer prior to arriving at the register cache. The topmost value corresponding to p5 has no remaining uses after bypassing, and it is therefore not written into the cache. The other value (in p8) is placed into the cache along with the number of expected remaining uses (one). This insertion requires the replacement of another entry and the one with the minimum number of uses (zero in this case, corresponding

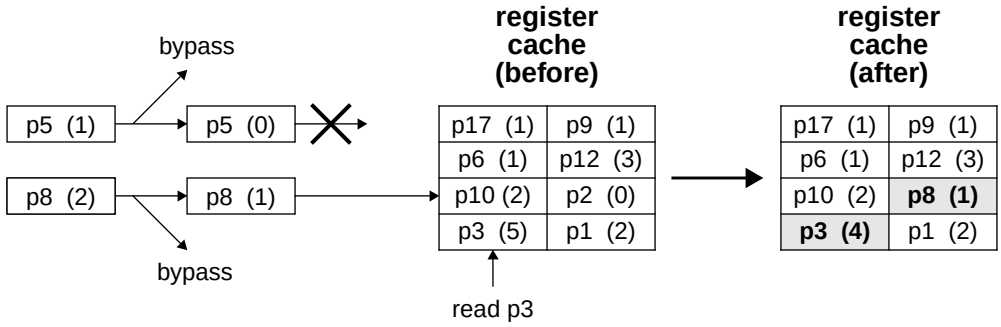


Figure 5.3. Use-based register cache management

to p2) is selected as the victim. The updating of a stored use count by a read of p3 is also illustrated. The shaded entries indicate changed entries in the register cache after the insertion and read.

The next section details the operation of a generic register cache without reference to its content management policies. Section 5.3 details register cache management via novel use-based insertion and replacement policies. The new register cache is evaluated and compared to previous register caching proposals in Section 5.4. Related work on register file optimizations is presented in Section 5.5, and Section 5.6 concludes.

5.2 Register Cache Operation

Figure 5.4 depicts pipeline diagrams illustrating the relationships among dependent instructions from issue through writeback. Each row represents the steps in the processing of a single dynamic instruction. Time is indicated by the cycle number above each column of the diagram. Therefore, a column does not correspond to a single hardware pipeline stage, but to all of the operations occurring in different stages at the same time. Inter-instruction value communication is indicated using arrows: operands are communicated through either the bypass network (dotted arrows) or storage (solid arrows), whether the register file or a register cache.

Figure 5.4(a) shows the operation of a processor with a three-cycle (read and write) register file and no register cache. I2-I6 are all data-dependent on I1 (only). I2-I5 receive their input via a four-stage bypass network. This bypass network is insufficient to completely hide the register file latency; thus, instructions dependent on I1 may not issue in cycles 6 or 7 (indicated by an $\times$). I6, which issues in cycle 8, can finally obtain I1's result value from the register file. In this example,

a full bypass network would require six stages. In an eight-way superscalar machine, this means each ALU input (as many as 16 in an eight-way machine) must choose among 49 input sources ($6 \times 8 = 48$ possible bypassed values + 1 from the register file). Even the four-stage network shown requires a 1-of-33 selection.

The operation of a register cache in a similar pipeline appears in Figure 5.4(b). In this diagram, I2-I4 are data-dependent on I1. I4 appears twice (I4a and I4b) to illustrate the cases in which a needed input is present in or absent from the cache. I5 is data-dependent on I4b. Adding a register cache allows for a smaller bypass network (covering only the cache itself) and reduces the read latency for most instructions (I1-I4a). In effect, the register cache takes the place of the register file, providing the access bandwidth required by the execution core. Each instruction implicitly assumes that its inputs reside in the register cache, which is accessed in the cycle after an instruction is issued (e.g., cycle 3 for I2). Instructions issuing with and after I4 cannot obtain I1's result value through the bypass network and must obtain it from the register cache. I4a shows the normal case—a *register cache hit*—in which the value is present in the cache. If the value from I1 were not present in the cache, a *register cache miss* would result. I4b illustrates this case, with the detection of the miss indicated by a star.

In the event of a register cache miss, the input value must be obtained from a storage structure other than the cache. This structure is called the *backing file* (BK file or BK in the pipeline diagram), and it is basically the original full-sized register file in a recovery role. To ensure that no values are lost, all values must be written to the backing file. Therefore, it must be able to support the full write bandwidth of executing instructions. The writing of the backing file commences in parallel with the (optional) writing of the value to the register cache. Because the register cache and bypass network filter the vast majority of reads, however, in those rare instances in which a value must be obtained from the backing file, a single read port (which can be shared with one of the write ports) suffices.

The pipeline diagram assumes backing file read and write latencies of only two cycles, compared with the three-cycle latency for the register file of Figure 5.4(a). By virtue of the significantly lower number of ports compared to a register file—as little as one-third of the original number—a backing file will be smaller and faster than a register file even though they must have

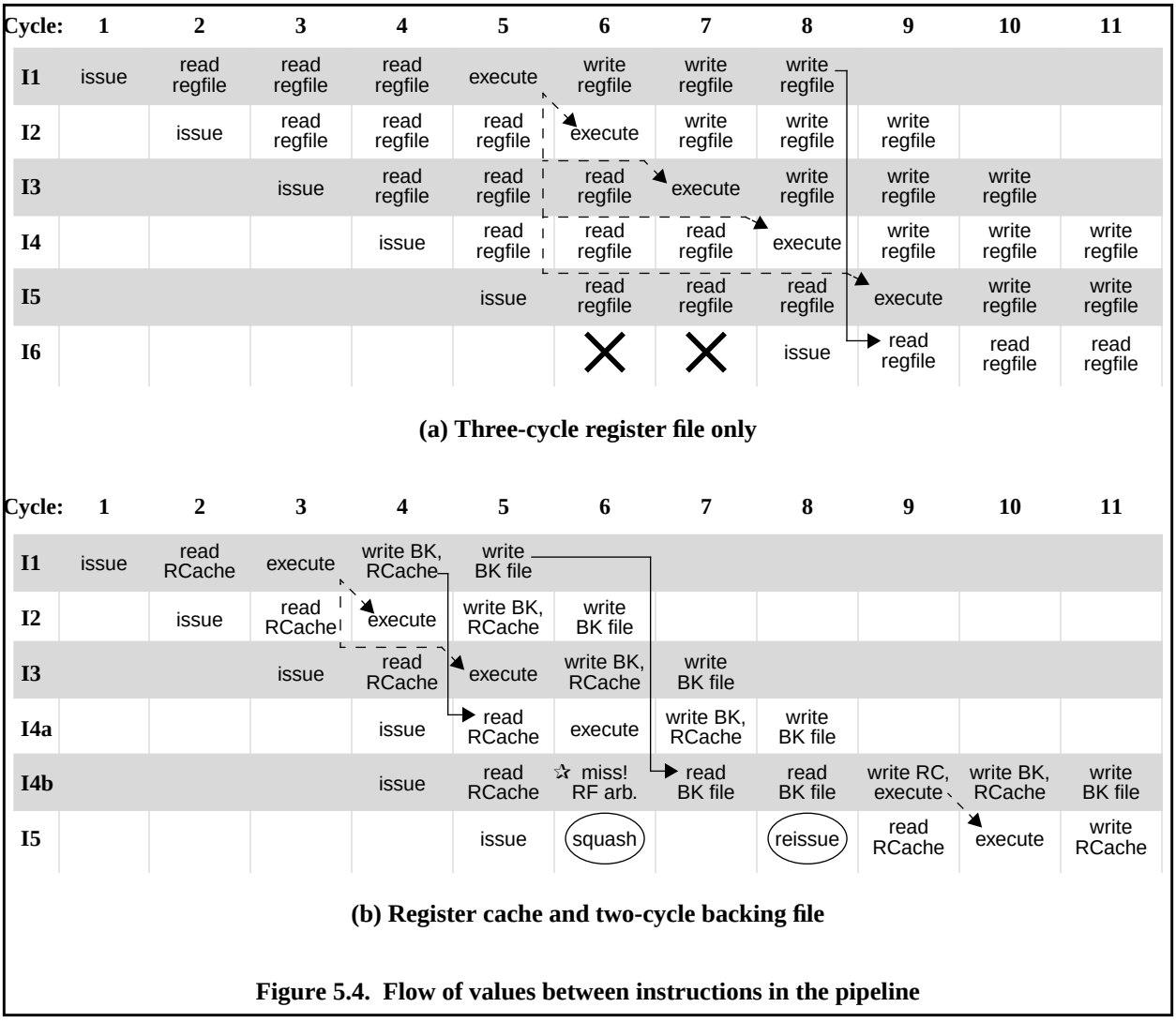


Figure 5.4. Flow of values between instructions in the pipeline

the same capacity. Unlike a register file, the latency of a backing file is not critical, being exposed only during a register cache miss.

Referring again to Figure 5.4(b), note that by the time a miss is detected, subsequent dependent operations (e.g., I5) may have already issued, speculatively assuming their parent would find its own inputs in the register cache. When this assumption fails, these instructions must either stall until their parent completes or replay (i.e., reissue at a later time). This situation is exactly analogous to the effect of a data cache miss under load-hit speculation [72, 90]. Stalling the dependent instructions is difficult because the issue pipelines must buffer them while allowing other, independent instructions to pass them. Replay-based solutions are also complicated

although several different processors have already implemented them to support load-hit speculation (e.g., the Alpha 21264 [48] and the Intel Pentium 4 [40]).

A register cache miss results in the replay of *all* instructions—dependent or not—issuing in the cycle after the missing instruction issues (equivalent to the model implemented by the Alpha 21264). Instructions independent of the missing instruction may then reissue, while the dependent instructions are delayed. In the example of the figure, when the miss is detected at the beginning of cycle 6, all of the instructions issued in the prior cycle (e.g., I5) are squashed. The miss signal also blocks the issue of any instructions dependent on the miss (or any of the squashed instructions) occurring at the end of cycle 6. Independent instructions that were squashed may reissue during cycle 7. Instructions dependent on I4b become eligible for reissue as the backing file read finishes (cycle 8), and they obtain their input value from the bypass network at the beginning of cycle 10.

The delay experienced by an instruction that misses in the register cache can vary if there is contention for the lone backing file read port. The handling of a register cache miss includes a cycle to arbitrate for this shared resource. If multiple register cache misses occur in the same cycle, the arbiter will delay the resolution of the misses such that only one backing file read occurs per cycle. For long backing file latencies and small bypass networks, a register cache miss can require a value that has not yet finished writing to the backing file, requiring an additional bypass network on the backing file read port to bypass incomplete writes.†

The issue port used for the cache-missing instruction is also blocked until the miss is resolved in order to prevent contention at the functional unit between the resolving miss and subsequently-issued instructions. Otherwise, complicated mechanisms would be required to handle potential reordering of instructions between issue and completion. Blocking the issue port also has the side effect of guaranteeing that the register cache write port will be free by the time the missed value is

---

† The conditions under which a backing file bypass is needed can be illustrated by considering adding additional backing file write stages to I1 in Figure 5.4(b) until the write no longer completes before the backing file read by instruction I4b. Changing the number of overall bypass stages affects the earliest instruction that can experience a register cache miss. For example, adding a third bypass stage would mean that I4b could not miss, delaying the first possible backing file read for I1's result to cycle 8. The number of backing file bypass stages that are needed equals $L_{bf,write} - N_{bypass} - 2$, where $L_{bf,write}$ is the write latency of the backing file and N_{bypass} is the number of bypass stages. Where this quantity is less than or equal to zero, no bypassing of the backing file is required.

retrieved from the backing file. In parallel with the resumed execution of the instruction experiencing the miss (e.g., during cycle 9 for instruction I4b in the figure), the value is placed into the register cache using this write port to avoid subsequent misses on that value. This operation is referred to as a *register cache fill*.

5.3 Use-Based Register Cache Management

The high cost of register cache misses means that minimizing their occurrence is crucial to realizing the performance benefit of a register cache. For a given cache capacity and organization, the miss rate will be a function of the insertion and replacement policies. In defining such policies, the main consideration will be ensuring that the limited cache space contains the proper values—namely, those values yet to be read by unexecuted consumer instructions.

Use-based register caching differs from previous register caching proposals by using the information provided by degree of use prediction to identify these values. Degree of use prediction provides the number of times that a result value will be needed; by monitoring the uses of that value as they occur, the number of *remaining uses* can be determined. The use-based insertion and replacement policies exploit the availability of remaining use information to keep the proper values in the cache.

The role of the insertion policy is to filter values that have no uses left after bypassing. Figure 5.2 illustrated the substantial role of the bypass network in value communication. By accounting for these bypasses, values that have reached all of their consumers need never pollute the cache, avoiding the possibility of evicting a still-live value stored there. Each time a value is bypassed, its degree of use is decremented. When the value must be written to the register cache, the write is blocked if the adjusted degree of use is zero; otherwise, the value is written into the cache along with the number of uses remaining, which enables use-based replacement.

When empty cache entries are not available to handle an insertion, the replacement policy is invoked to select a victim.† The replacement of a valid cache entry does not necessarily imply the eviction of a live value (which would lead to a subsequent register cache miss). Once a live value

† Empty cache entries arise because cache entries must be invalidated when their associated physical register tag is freed. Otherwise, the cached value could be supplied incorrectly to a subsequent instruction assigned the reclaimed tag.

is cached, the cache supplies the subsequent consumers. At some point, all of the consumers of the cached value are satisfied and the value is dead. As indicated by Figure 5.1, many values in a register file are in precisely this state. The goal of the replacement policy is to select such a value for replacement. The remaining use count stored with each cached value facilitates this choice. Similar to the adjustments of the remaining use count occurring within the bypass network, the use counts stored within the cache are decremented as their associated values are read. Use-based replacement simply selects a victim with the fewest remaining uses.

The rest of this section details use-based register cache management. The insertion and replacement policies are described in more detail in Section 5.3.1 and Section 5.3.2, respectively. Tracking the number of remaining uses for each value is central to the scheme and is the topic of Section 5.3.3. Section 5.3.4 covers the implications of incorrect use information.

5.3.1 Register cache insertion policy

The register cache insertion policy seeks to avoid caching values that will never subsequently be read. A prerequisite (assuming all instructions are useful) is the existence of an alternative value communication mechanism—in this case, the bypass network. The bypass network is ideally-suited for the direct communication of a value to consumers issuing within a short window after the value becomes available. Since the availability of a new value leads to the scheduling of operations waiting on that value, many of a value's consumers issue within this window and receive the value from the bypass network. Values with low degrees of use may reach all of their consumers in this manner. Because the insertion policy prevents such values from entering the cache, it is best described as *use-based filtering*.

Use-based filtering is similar to a heuristic proposed by Cruz et al. [24] labeled *non-bypass*, which writes a value into the register file cache only if it was not bypassed to any instructions prior to the write. In effect, this scheme uses bypassing as a rough proxy for the number of remaining uses. Since most values have a single consumer, the intent is to keep these values from polluting the register cache when their consumers are satisfied from the bypass network. However, values with many consumers that bypass to only some of their consumers prior to the write are also filtered from the cache, resulting in additional misses. The non-bypass heuristic also leads to the writing of all useless values into the limited register cache since, by definition, they

will not bypass to any consumers. Figure 4.3 shows that a substantial number of needless writes may result. Use-based filtering avoids the caching of useless values detected by the degree of use predictor.

Filtering values from the register cache based on how they are bypassed requires the ability to detect bypass communication before the cache write takes place. Communication within the bypass network occurs via matching of the input physical register tag of an issuing instruction with the destination physical register tag of a recently-generated value. This dependence detection operation occurs in parallel with the access of the register cache for the same value. Therefore, the occurrence of a bypass is known at the end of the register cache read stage of the instruction receiving the value. In order to influence the writing of that value, then, the write must occur after that point.

Consider the instruction I1 in the pipeline diagram of Figure 5.4(b). It writes its result into the register cache during cycle 4. The first consumers of its result, however, issue at the end of cycle 2. Therefore, during cycle 3, bypasses can be detected and used to update the remaining-use count for the value, initially set by the degree of use predictor. If these bypasses comprise all of the predicted uses of I1's result, the cache write during cycle 4 may be avoided. Otherwise, the value and its remaining uses must be written into the cache. Note that potential consumers of a value that issue two cycles after the value's producer (e.g., I3) also obtain their inputs from the bypass network but cannot affect the writing of the register cache. These instructions will be in the cache read stage while their parents are in the cache write stage, requiring the bypass network to forward the communicated values as before. However, the input register tags of these instructions are not available before they are in the cache read stage, and, by this time, the parent instruction will have already commenced writing the register cache.

These *missing bypasses* lead to inflated remaining-use counts being stored in the cache.† The consequences of these inflated use counts are addressed in Section 5.3.4. Note that by delaying the writing of the register cache, it is possible to account for more total bypasses. In Figure 5.4(b), if the register cache write for I1 were delayed one cycle to cycle 5, bypasses to I2 and I3 could gate the writing of I1's result. However, an additional bypass stage would also be

† In the initial work on use-based caching [17], it was assumed that missing bypasses updated the cache later, but such a design is probably not realizable.

required since I4 would otherwise not be able to obtain I1's result from either the cache or the original bypass network. Overall, the final stage of bypassing always accounts for the missing bypasses. Fortunately, the data of Figure 5.2 indicate that first stage bypasses are the most important.†

5.3.2 Register cache replacement policy

Previous register cache proposals have assumed LRU [24, 92] or FIFO [11] replacement, neither of which are particularly suited to the behavior of register values. Due to the dominance of values with few uses, any given use of a value is probably its last.‡ The LRU scheme, however, makes recently-used values the *least* likely to be replaced. FIFO replacement ignores uses altogether, always selecting the oldest entry as the victim. While values with a high degree of use are rare, they have long lifetimes (Figure 2.4) and account for many of the input values likely to remain after bypassing (Figure 2.3). Each such value can cause multiple misses as it is repeatedly written and eventually replaced from the cache. The availability of future use knowledge (in the form of remaining-use counts associated with each cached value), however, allows for more intelligent victim selection.

To minimize the number of register cache misses, use-based replacement selects the cache entry with the smallest number of remaining uses as the victim. In the event of a tie, the oldest entry is selected (i.e., FIFO). Most of the time, victims selected in this manner have zero remaining uses, and the evictions do not result in a future cache miss. This single reason accounts for the superiority of this method over either LRU or FIFO replacement: known-dead values are replaced preferentially. For victims with uses remaining, one or more future misses on the replaced value are likely.

---

† While the data in the figure pertain to a six-stage bypass network, the dominance of the first stage bypasses holds across bypass networks of different sizes. For bypass networks of two, four, six, and eight stages (corresponding to register file latencies from one to four cycles), first stage bypasses account for 82.2%, 68.7%, 63.4%, and 60.4% of all bypassed values, respectively.

‡ The probability that a given use of a value is its last can be calculated using the analytical model from Section 2.5.1. Given a value with a degree of use $x > 0$, the probability that a use is the last is $1 / x$. Summing this probability over all degrees of use weighted by their frequencies of occurrence gives:

$$P = \sum_{x=1}^{\infty} \frac{1}{x} P[D = x] = \sum_{x=1}^{\infty} \alpha \cdot x^{-\beta-1} = \alpha \cdot \zeta(\beta + 1)$$

Using the values $\alpha = 0.717$ and $\beta = 2.55$ from Table 2.4 gives the likelihood of any use being a value's last as 80.4%.

The importance of evicting the value with the fewest remaining uses is partially due to the delay imposed upon operations that need that value after it has been evicted. A miss on a high-use value can delay more operations than a miss on a single-use value. In the latter case, there is a greater likelihood of other independent instructions being able to execute to partially hide the cost of the miss. If, however, the evicted value is the parent of many instructions (e.g., the base address of a structure in which many fields are accessed), then it is possible that many or all of the ready instructions will experience the full latency of the miss.

Another reason that selecting the victim with the fewest uses is preferred is that it helps reduce the number of future misses possible on the same value. Even though the register cache is filled on a miss (Section 5.2), a filled value makes a good eviction candidate because its use count is cleared. Remaining use counts are only kept for values in the register cache (and in the bypass network prior to their arrival). This avoids the complexity associated with writing this information somewhere else when an entry is replaced. Therefore, when a value is brought back into the register cache after a miss, the use count is lost and assumed to be zero (see the discussion of the fill default in Section 5.3.3); thus, the greater the number of remaining uses a value has when evicted, the more misses it can cause.

The identification of the victim under the fewest-use replacement policy is a source of complexity, especially in highly-associative register cache organizations. Fortunately, simpler approximations of this policy are possible. The main shortcoming of FIFO replacement is the potential for multiple misses on certain high-use values. A slight modification of the FIFO replacement policy would skip over values with more than a threshold number of uses. Another hybrid is possible for addressing the main problem with LRU replacement. A modified LRU could be implemented in which any entry with zero remaining uses would take precedence over the nominal LRU during victim selection. In both of these hybrid schemes, the availability of the use information enables improvement of the original replacement algorithm.

5.3.3 Counting remaining uses

The use-based policies just described depend upon the availability of a remaining-use count for each value, which originates from the degree of use predictor. Once initialized, each count must be updated by uses of the associated value. Within the bypass logic, matches on each result tag

configure the bypass multiplexors. A tag match implies a use by a soon-to-execute instruction. To implement use-based filtering, additional circuitry accumulates the number of these matches occurring for each value within the bypass network (matches in the last bypass stage are missing bypasses and are not counted; see Section 5.3.1). This number is then subtracted from the degree of use prediction to implement the insertion policy. After the adjusted use count and the value are present in the cache, the counts are updated by subsequent reads of the value.

Use counts equal to the maximum predictable degree of use (Section 3.2.1) are handled differently. Recall that a degree of use predictor is saturating: it uses the maximum representable number of uses to denote that and all higher numbers of uses. This presents an interesting problem with regard to managing the register cache. If a single value will have millions of uses, the cost of repeatedly evicting that value will be very large. Therefore, subtracting uses from the saturated maximum is not the desired behavior. Instead, the remaining-use count is not updated for values with the maximum predictable degree of use, effectively pinning such values in the cache until the corresponding physical register is freed; of course, bypasses must not adjust a saturated use count either.†

The fact that a portion of values are pinned in the cache based on their predicted degree of use has implications for the choice of the maximum predictable degree of use. It is desirable to pin the smallest possible number of values in the cache, which favors a higher degree of use limit. However, higher maximums have a hardware cost in the degree of use predictor, the register cache, and the associated data paths for tracking and accumulating uses. Of these, the complexity of the logic for updating use counts is the most critical. Updating remaining use counts of more than a few bits is likely to be prohibitively difficult. This situation is well-suited for prediction grouping as described in Section 3.2.4. For example, instead of using two bits to represent degrees of use of 0, 1, 2, and >= 3, a better encoding in this application would be 0, 1, 2 to 7, and >= 8, reducing the number of pinned values substantially at the cost of some inaccuracy in the use counts.

---

† Note that no values are actually pinned in the cache—their use counts are simply not decreased from the maximum, making it much less likely that they will be replaced. If, during an insertion, a cache set contains only values having the maximum use count, one will be selected as a victim, independent of the use count of the incoming value.

Less than perfect predictor coverage leads to the inability to initialize some remaining use counts. For these values, an implicit prediction (see Section 3.2.3) called the *unknown default* is assigned. The choice of this default is dictated by the capacity pressure on the cache. A value with a higher default is more likely to end up in the cache after bypassing to all of its actual consumers—perhaps resulting in the eviction of a more useful value (the consequences of dead values in the cache is the topic of Section 5.3.4). When space is abundant, it is better to place them into the cache by default to avoid the possibility of a miss. The data in Section 5.4 indicate that capacity pressure is extremely important for reasonable register cache sizes, so an unknown default of one is used. Note that this default leads to behavior identical to the non-bypass insertion policy (see Section 5.3.1) for these values.

A similar situation arises after a register cache fill because the backing file does not contain use information. As in the case of an unknown initial degree of use, the remaining-use count is set to an algorithm parameter called the *fill default*. Again, noting that any use of a value is likely the last (see Section 5.3.2), it is desirable that those values with known real uses be given priority in avoiding replacement. Therefore, a fill default of zero is assumed. Note that it is still important to perform the fill since a cached value will supply consumers regardless of its remaining use count. So long as there is not any contention that would lead to replacement of the filled value, it can reside in the cache for some time, even with zero remaining uses.

5.3.4 Incorrect use information

Inaccurate remaining-use counts arise from degree of use mispredictions, the use of unknown and fill defaults, missing bypasses, and the counting of wrong-path uses resulting from control-flow speculation (e.g., branch prediction). These events result in disagreements between the number of remaining uses recorded in the register cache and the number actually outstanding. Incorrect use counts never lead to incorrect operation: regardless of the contents of the register cache, all values are available from the backing file. However, they can result in poor performance by affecting the ability of the use-based policies to keep live values within the register cache.

Incorrect remaining-use counts manifest in one of two ways. First, a value might be present in the register cache with predicted remaining uses that will never be observed. These are referred to as *stale values* and are exactly analogous to those registers in the register file that contain dead

values (Figure 5.1). The storage of stale values inflates the number of register cache entries required and can result in the eviction of genuinely useful values. Alternatively, the cache state could indicate that a value has no remaining uses even though that value is still live. These *falsely-dead* values can lead to a register cache miss if the value is evicted before their outstanding uses are satisfied.

The impact of stale values is limited by two factors. Most importantly, the invalidation of register cache entries when the corresponding physical registers are freed (necessary to ensure correctness) bounds the lifetime of stale values in the register cache. Also, stale values are not immune from the fewest-remaining-use replacement policy. Like all other values, stale values are likely to have a small number of uses, especially once any actual uses have been counted. Therefore, they are at least as likely to be selected as a victim as a live value with actual uses remaining.

The potential cost of falsely-dead values is also mitigated in practice for two reasons. First, values remain in the cache—even if their remaining-use count reaches zero—until they are explicitly chosen as a victim by the replacement policy. Thus, unless there is actual contention among live values for entries in same set as the falsely-dead value, the cache will continue to supply the value to consumers. Second, for many values, all consumer instructions will obtain their inputs from the bypass network. Therefore, especially for values with few uses, all of those uses may be satisfied without incurring a register cache miss, even if the predicted number of uses was too low.

5.4 Evaluation

This section presents an evaluation of use-based cache management policies. The processor model in which the register caches are evaluated is described in Section 5.4.1. Section 5.4.2 addresses the capacity and organization of the register cache. Section 5.4.3 discusses the costs of register cache misses and their role in determining performance. Next, the different register cache policies are compared in isolation: Section 5.4.4 looks at the insertion policy while Section 5.4.5 examines the replacement policy. Section 5.4.6 revisits register cache misses, discussing the performance results of the prior sections in terms of different kinds of misses. Finally, Section 5.4.7 illustrates the sensitivity of the register cache to the cache size and machine width.

5.4.1 Processor model

The implementations for which a register cache is likely to be beneficial are wide-issue machines with deep pipelines. The combination of these two attributes creates the need for many physical registers, while the deep pipeline implies that access to a monolithic register file could extend over several pipeline stages. Therefore, it is important to evaluate register caching in such a machine. The modeled processor configuration outlined in Table 5.1 reflects this consideration. It is an eight-issue superscalar processor with a deep pipeline (16-cycle minimum to redirect fetch after a branch mis-speculation) supporting up to 320 in-flight instructions. The front end, execution resources, and cache hierarchy are similarly aggressive. The effect of using a more realizable superscalar width of four is examined in Section 5.4.7.

Especially relevant for the evaluation of register caching are the size and latency of the physical register file and backing file and the structure of the bypass network. The physical register file contains 320 registers (320 in-flight instructions $\times$ 80% value-generating instructions + 64 architectural registers). The register file latency only affects the baseline performance against which register caching is evaluated. Read and write latencies are each set at three cycles, similar to the

Table 5.1: Simulated Processor Parameters

Pipeline	8-wide superscalar; 5-stage fetch (next address + I-cache access + fetch queue), 2-stage each decode and rename, 1-stage dispatch (write into window), issue, and commit. 16-cycle minimum fetch redirection on branch mis-speculation.
Front end	Up to 8 non-nop instructions per cycle from up to 2 cache lines. Each fetch block can contain up to one taken branch and any number of untaken branches. 48-entry instruction queue between L1 I-cache and decode.
Issue/Execute	200-entry scheduling window, oldest ready first. 320-entry reorder buffer. Eight issue ports: (1) simple integer (no mult), (2,3) load or store, (4,5) simple integer or simple FP (no mult/div/sqrt/branch), (6,7) any integer/FP incl. branch, (8) simple integer or load or store.
Register/ Bypass	320-entry physical register file, 3-cycle latency OR 1-cycle register cache and 320-entry backing file, 2-cycle latency. 2-stage bypass network.
Memory	32KB, 2-way set-associative L1 instruction and data caches with 64-byte blocks. 2MB, 4-way set-associative unified L2 cache with 128-byte blocks, 12-cycle latency. 160-cycle memory latency. 128-entry load queue and 128-entry store queue.
Degree of Use Predictor	8K-entry, 8-way set-associative, 13.4KB predictor described on page 80.

example of Figure 5.4(a). Performance results in this section are presented as speedups relative to this baseline multi-cycle register file.

The backing file is the same size as the physical register file it replaces (320 entries here). The latency of the backing file affects the performance of register caching via the miss penalty. The backing file does not need to support the read bandwidth of a register file since it is only read on register cache misses. Therefore, its access latency can be lower than the physical register file; here a two-cycle latency is assumed.

Any implementation will include the largest bypass network allowed by the design constraints (until the full register file latency is covered). Because a large bypass network represents a significant limiter to clock frequency scaling [66], in the eight-wide processor model considered here, a two-stage bypass network is assumed. Such a bypass network represents full bypassing for the register cache, but limited bypassing [2] for the three-cycle register file.

5.4.2 Register cache size

Based on the number of live values indicated in Figure 5.1, the register cache should have at least 60 cache entries to have a hope of containing all of the live values. The size of a register cache will ultimately be determined by the need to maintain single-cycle access. Since the base case for performance comparison assumes a three-cycle latency for a 320-entry register file, a single cycle register cache should certainly have no more than about 106 entries (320 ÷ 3). However, while the register file is direct-mapped, a register cache needs to be associative to offer reasonable performance [17], which will make it slower for the same number of entries.

Given the three-cycle latency of the original physical register file, a single-cycle cache of up to 80 entries is probably reasonable depending on the associativity. The latency penalty of a fully-associative design as well as the difficulty of implementing a global replacement policy over so many entries indicate that a set-associative design is more realistic. In the remainder of this chapter, three specific design points will be evaluated: 64×4, 80×4, and 80×8, where $m \times n$ indicates an m-entry, n-way set-associative design.

Going from fully-associative to set-associative introduces conflict misses, which generally will prevent the realization of the full performance potential of a given capacity. These can be mitigated somewhat by using decoupled indexing [17]. Decoupled indexing explicitly assigns a

register cache set to each physical register as it is allocated instead of relying on the implicit derivation of the set index from the register tag. Consumers of the register are provided with its cache set via the standard renaming process; however, instead of receiving only a physical register tag, they receive a tag and a register cache set index. Set indices are assigned using a policy that attempts to minimize conflicts within the register cache. A very simple policy that works well assigns set indices in a round-robin manner, with each newly allocated physical register assigned to the next set sequentially. Since execution order often resembles rename order (due to data dependencies), this policy helps to keep values produced within a short period of time in different cache sets. This indexing policy is used for all of the set-associative designs presented in the remainder of the chapter.

5.4.3 Misses

The most important determinant of the performance of any register caching scheme is the aggregate cost of the register cache misses. This cost is a combination of the penalty of each miss and the frequency of their occurrence (i.e., the miss rate). The per-miss penalty is independent of the policies and organization of the register cache. Instead, it depends on the miss model and the particular benchmark. The miss model, which includes the backing file access, the issue port stall, and the replay of operations issuing in the register cache miss shadow, merely defines the costs associated with each.

A given benchmark may be more or less sensitive to misses due to specific interactions with components of the miss model. For example, a benchmark with a high average IPC will be more sensitive to misses because more independent computation will be delayed. Similarly, a benchmark with a high proportion of instructions requiring a limited execution resource (e.g., a floating point divider) may perform especially poorly as misses lead to the temporary unavailability of that resource. Where a benchmark's performance is severely limited by other bottlenecks (e.g., L2 misses), the benchmark may be relatively insensitive to register cache misses.

How the miss rate itself affects performance is more straightforward: more misses equals lower performance. This relationship is illustrated quite clearly in the data of Figure 5.5. Each point corresponds to a different combination of cache size, associativity, and insertion and replacement policies. Note that there is no significance to the division of the benchmarks among

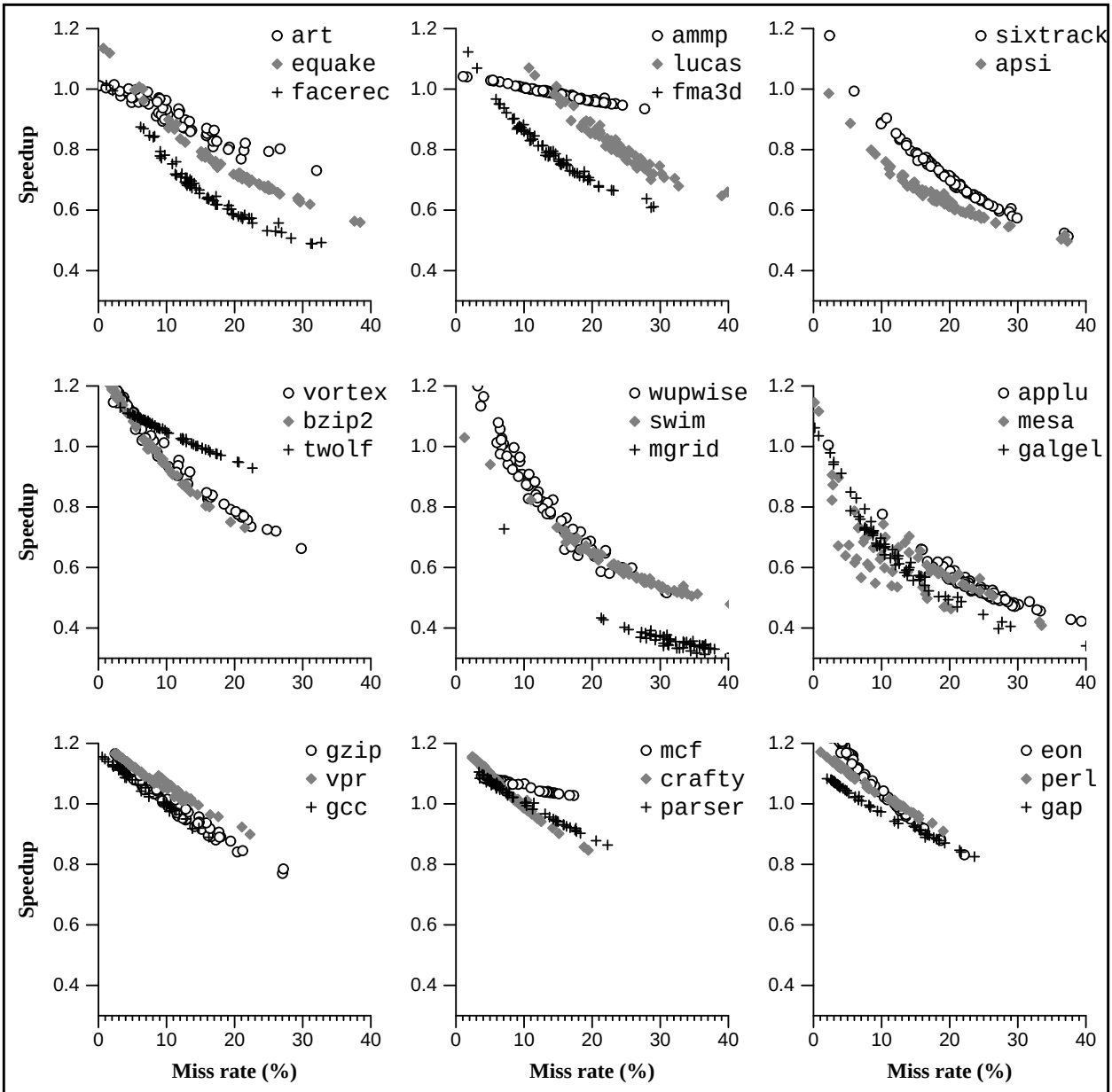


Figure 5.5. Effect of miss rate on performance

the different graphs other than clarity of presentation. With few exceptions (e.g., `ammp` and `mcf`), the data show a strong negative correlation between performance and miss rate. The data generally falls on well-defined curves regardless of the different cache parameters. Where scatter occurs (e.g., `wupwise`, and to a lesser extent, `mesa` and `art`), the overall correlation is still readily visible. The slope and position of a curve fitting the data for a given benchmark offer a wealth of information.

The slope of the curve directly indicates the per-miss penalty. As discussed previously, this depends on the peculiars of the miss model and the benchmark. Since the miss model is the same for all of the experiments represented here, variations in the miss penalties (slope) of the different curves must indicate attributes of the benchmarks themselves. For example, `ammp`, which spends most of its time waiting on memory, shows very little sensitivity to the miss rate. Conversely, among the integer benchmarks, `bzip2` shows the largest miss penalty because of a high base IPC. The curvature seen among many of the benchmarks is related to the interaction of the miss penalty with the base IPC. At high miss rates, the cost of each miss goes down as they cease to delay as much independent useful work.

The vertical position of a given plot shows the benefit of using a register cache over a multi-cycle register file. Consider the intercept of a curve with the y-axis (speedup axis): that point indicates the performance advantage of a perfect register cache. For example, the performance of `wupwise` with a perfect register cache is over 20% better than its performance with the three-cycle register file; `art` on the other hand, does not show much improvement even for miss rates near zero.

Clearly, any performance advantage offered by one register caching algorithm over another will result from a decrease in the miss rate. For a fixed size and associativity, then, the miss rate must be linked solely to the predictor policies. This observation was offered unsubstantiated at the beginning of Section 5.3 when considering the attributes of a successful cache management policy.

In examining the effects of the prediction policies on performance in the following sections, it will be helpful to define two possible kinds of misses. *Filtering misses* are the result of an attempt to access a value that was not put into the cache because of the insertion policy. *Eviction misses* result from the replacement of a live value in the register cache. Eviction misses may be due to a poor choice by the replacement policy or they may simply occur because of capacity constraints or conflicts. Therefore, while filtering misses are solely attributable to the insertion policy, eviction misses depend on both the replacement and insertion policies since the insertion policy can mitigate capacity pressure on the cache via write filtering.

5.4.4 Comparing insertion policies

Figure 5.6 compares three different cache insertion policies. For each policy, the dark gray bars show the performance of the 64×4 cache, the light gray bars the 80×4 cache, and the black bars the 80×8 cache. The A bars correspond to a policy that writes all result values into the register cache (i.e., no filtering is performed). The N bars show the performance of the non-bypass insertion policy [24] in which values that bypass to any number of consumers are not written into the cache. Note that the non-bypass policy still suffers from missing bypasses (see Section 5.3.1). The performance of use-based insertion filtering is indicated by the U bars. LRU is used as the replacement policy for all three configurations.

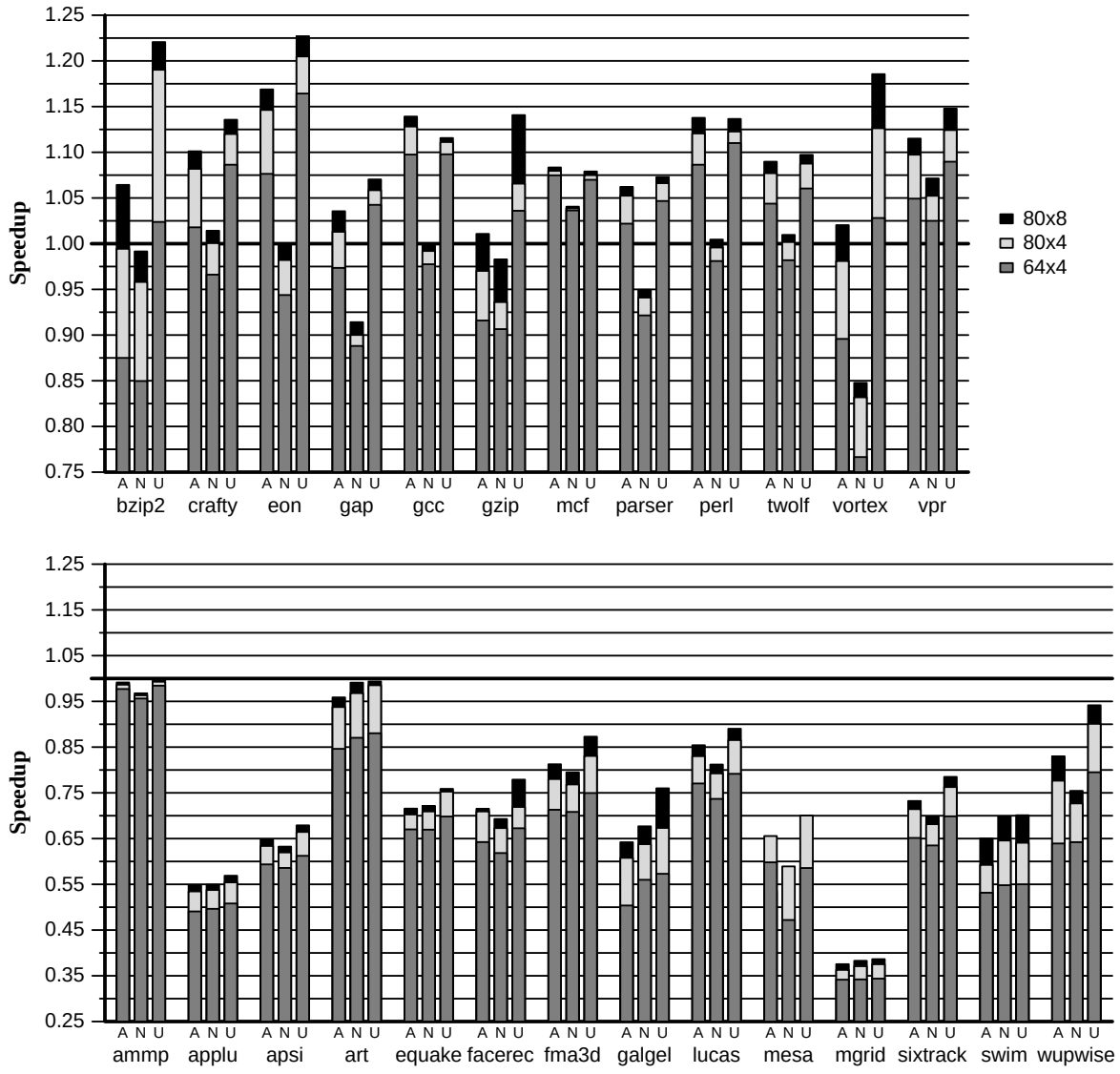


Figure 5.6. Insertion policies

The performance advantage of use-based filtering is substantial, especially where capacity limitations are important. Use-based filtering reduces the number of values written to the cache, requiring fewer replacements and relieving capacity pressure. This advantage is best illustrated in the floating-point benchmarks, where use-based filtering delivers the highest performance on every benchmark even for the 80×8 organization. Among the integer benchmarks, the policy still delivers the highest performance overall, although the policy that indiscriminately writes all values (A bars) can perform slightly better for some benchmarks, especially at large cache sizes. In these cases, capacity is not a limitation (Figure 5.10 shows that 64-80 entries suffice for the integer benchmarks); thus, the small advantage of reducing the number of writes is offset by the additional filtering misses on values that should have been cached.

Note that the non-bypass scheme universally performs worse than filtering based on use information. While single-use values account for more values than any other class, the combined number of all values with higher degrees of use is substantial (nearly 30%). Thus, there will be a significant number of values that bypass to some consumer yet still have additional consumers after bypassing. While use-based caching detects that additional uses are outstanding, the non-bypass scheme will filter these values, leading to additional filtering misses. Still, where capacity limitations are especially important (e.g., `galgel`), even low quality filtering is better than none at all, and the non-bypass scheme outperforms writing all values.

Table 5.2 presents additional data on the efficacy of use-based filtering for the 80×4 register cache (i.e., corresponding to the light gray bars in Figure 5.6) in terms of the percentage of values avoiding the cache (initially and completely) and the percentage written to the cache that are not subsequently read. Only the last of these is meaningful for the policy that writes all values (since the filtering percentages would be zero). Differences between the initial filtering percentage and the complete filtering percentage (i.e., the percentage of values *never* entering the cache) result from both poor filtering decisions and the difference in the sets of values under consideration. Poor filtering results in values eventually entering the cache because of a subsequent miss; thus, poor filtering leads to an initial filtering rate greater than the never-cached percentage (e.g., for `ammp`). There is also a slight difference in the two percentages because the initial filtering rate is a percentage of all values written, while the never-cached percentage considers only values generated by instructions that retire.

Table 5.2: Evaluating Use-Based Filtering

	% values filtered initially		% values never cached		% cached but never read		
Benchmark	Use-based	Non-bypass	Use-based	Non-bypass	Use-based	Non-bypass	write All
eon	53.50	44.90	54.93	40.01	28.41	46.20	67.92
bzip2	65.88	62.87	66.66	60.27	40.96	47.90	76.82
crafty	59.71	47.14	59.95	44.53	32.17	51.85	74.13
gap	67.18	52.04	68.00	47.00	27.23	57.55	76.55
gcc	62.28	51.55	62.61	48.58	32.74	52.68	74.93
gzip	58.73	42.47	62.16	43.52	35.09	57.58	74.43
mcf	57.51	47.37	57.71	42.90	24.34	49.61	69.39
parser	56.20	54.18	57.96	51.60	39.66	49.97	74.71
perl	61.55	49.38	62.08	44.39	29.99	52.53	73.35
twolf	53.91	51.97	55.12	45.92	30.66	44.99	68.49
vortex	58.56	36.33	60.11	33.85	24.66	59.83	72.82
vpr	50.37	45.69	52.14	42.57	28.17	43.98	65.87
ammp	41.28	38.99	43.42	33.70	37.79	54.68	65.18
art	24.63	27.96	24.62	22.86	24.60	28.73	45.15
equake	37.85	37.05	38.06	31.90	38.84	48.16	63.85
mesa	48.25	37.67	53.51	33.37	31.15	53.51	67.91
applu	21.21	23.68	22.12	21.63	53.89	54.33	61.48
apsi	30.08	27.66	30.43	23.58	42.60	45.87	57.15
mgrid	12.54	14.72	12.76	13.36	61.17	50.07	55.54
sixtrack	35.47	32.24	36.75	28.78	39.90	50.86	63.91
swim	18.99	25.62	19.06	21.59	52.01	45.23	57.05
wupwise	44.03	40.83	45.92	37.97	36.07	45.18	64.62
facerec	30.78	30.04	31.32	24.61	42.87	34.38	53.19
fma3d	36.39	33.85	38.02	31.12	39.53	47.20	62.44
galgel	39.16	36.03	39.32	32.40	30.85	31.24	49.36
lucas	23.21	25.56	23.21	20.46	42.33	45.24	54.17
Integer	58.77	48.82	59.95	45.43	31.17	51.22	72.45
Floating Pt.	31.71	30.85	32.75	26.95	40.97	45.33	58.64
C/C++	53.58	45.47	54.94	41.69	31.65	49.98	69.47
Fortran	29.19	29.02	29.89	25.55	44.12	44.96	57.89
All	44.19	39.15	45.31	35.48	36.45	48.05	65.02

The use-based filtering policy keeps more values from entering the cache than the non-bypass policy. In the case of the integer benchmarks, the difference is dramatic: over 30% more values (60% versus 45%) never enter the cache under use-based filtering. Overall, the use-based policy keeps 44% of retired values from ever entering the cache.

The filtering percentages also underscore the vastly different behavior of integer and floating-point benchmarks. Nearly twice as many values are filtered from the register cache in the integer benchmarks, leading to much higher capacity demands by the floating-point benchmarks. The lower filtering rate in the floating-point benchmarks can be attributed partially to the smaller number of floating-point execution resources. Floating-point instructions with ready operands frequently wait for execution resources; by the time the instruction can execute, its inputs are no longer available on the bypass network, leading to a lower bypass rate. With fewer bypasses occurring, fewer insertions are avoided and capacity pressure on the cache is increased. This conclusion is supported by the average number of ready instructions—28 in the floating-point benchmarks versus 11 in the integer benchmarks—and the percentage of all reads satisfied by the first-stage of the bypass network—25% floating-point, 46% integer.†

The percentage of cached values that are never read also quantifies the success of the insertion filtering policy. Ideally, this percentage would be zero, indicating that only values that would eventually be read would be cached. However, this percentage is also subject to two factors unrelated to the insertion filtering. First, capacity pressure can result in the eviction of a useful value from the register cache (i.e., one that *would* have been read) prior to any reads occurring. Second, the fills that occur on a register cache miss can bring values into the cache that may not be read again. In spite of these effects, the advantage of use-based filtering over the other two insertion policies is clearly evident by this measure. Use-based filtering decreases the number of values needlessly placed into the register cache by 24% and 44% versus non-bypass filtering and no filtering, respectively.‡

---

† These data correspond to use-based insertion filtering and LRU replacement with an 80×4 register cache.

‡ (48.05% – 36.45%) ÷ 48.05% = 24% and (65.02% – 36.45%) ÷ 65.02% = 44%.

5.4.5 Comparing replacement policies

The performance of various replacement policies are compared in Figure 5.7. The presentation is identical to Figure 5.6 with 64×4, 80×4, and 80×8 register caches (dark gray, light gray, and black bars, respectively) evaluated under each policy. Use-based filtering is used as the insertion policy in all cases. Within each bar group, the replacement policies are: L(RU), F(IFO), and D(egree of use-based), which selects for replacement the entry with the fewest remaining uses.

Again, the advantage of employing a use-based policy is evident. As was the case with the use-based insertion policy, the benefit tends to be more pronounced under capacity constraints.

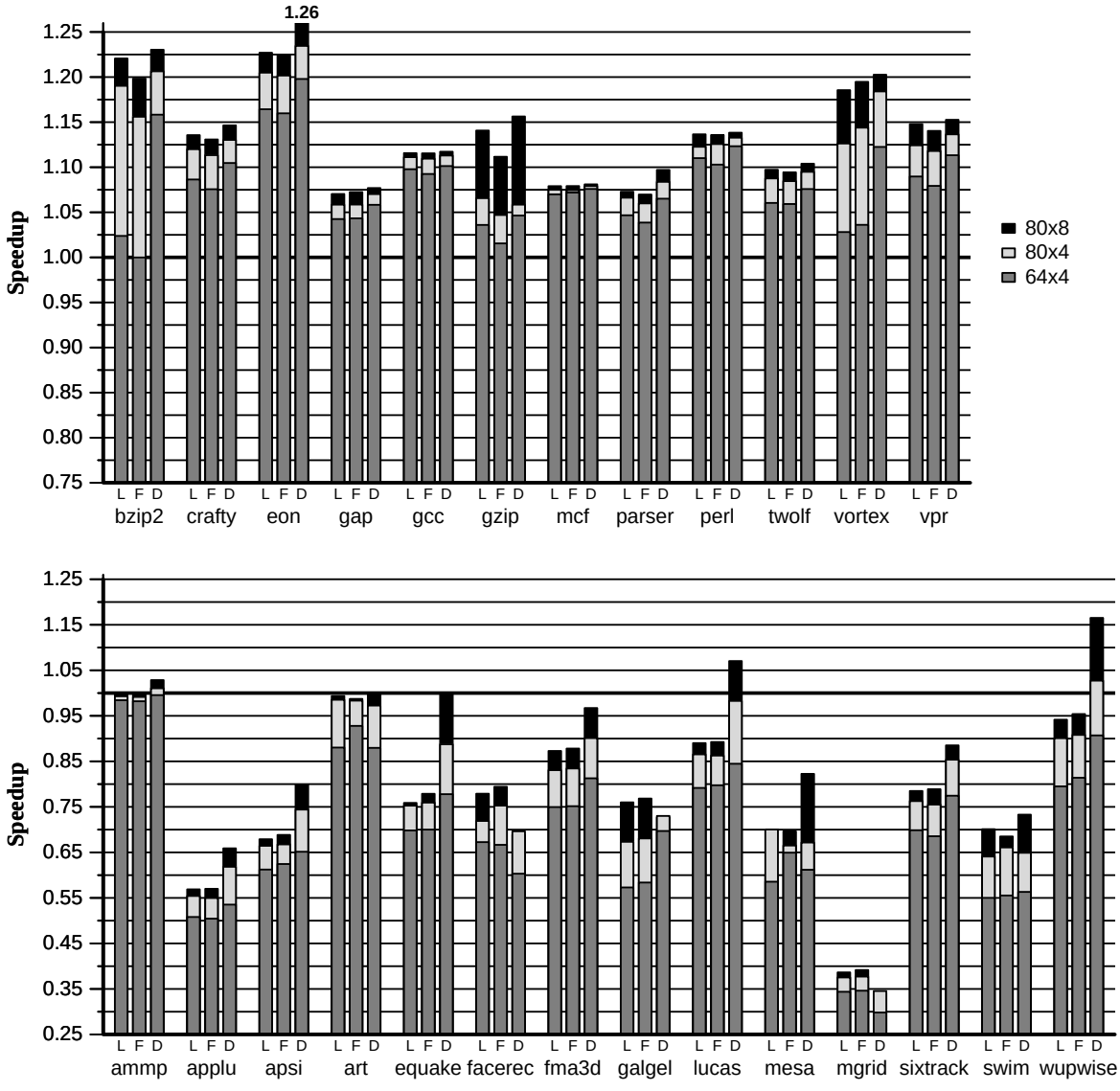


Figure 5.7. Replacement policies

The fact that the use-based policies work well for relatively small caches indicates that they do a good job of maintaining the most important set of values within the limited amount of space. Because the latency of a register cache depends on its size, a use-based cache can deliver equivalent performance to that of previously-proposed policies with a smaller, lower-latency organization.

Section 5.3.2 also proposed modifications of the LRU and FIFO policies based on the availability of use information. These hybrid schemes were meant to address some of the complexity surrounding the identification of the entry with the minimum degree of use. Figure 5.8 shows the

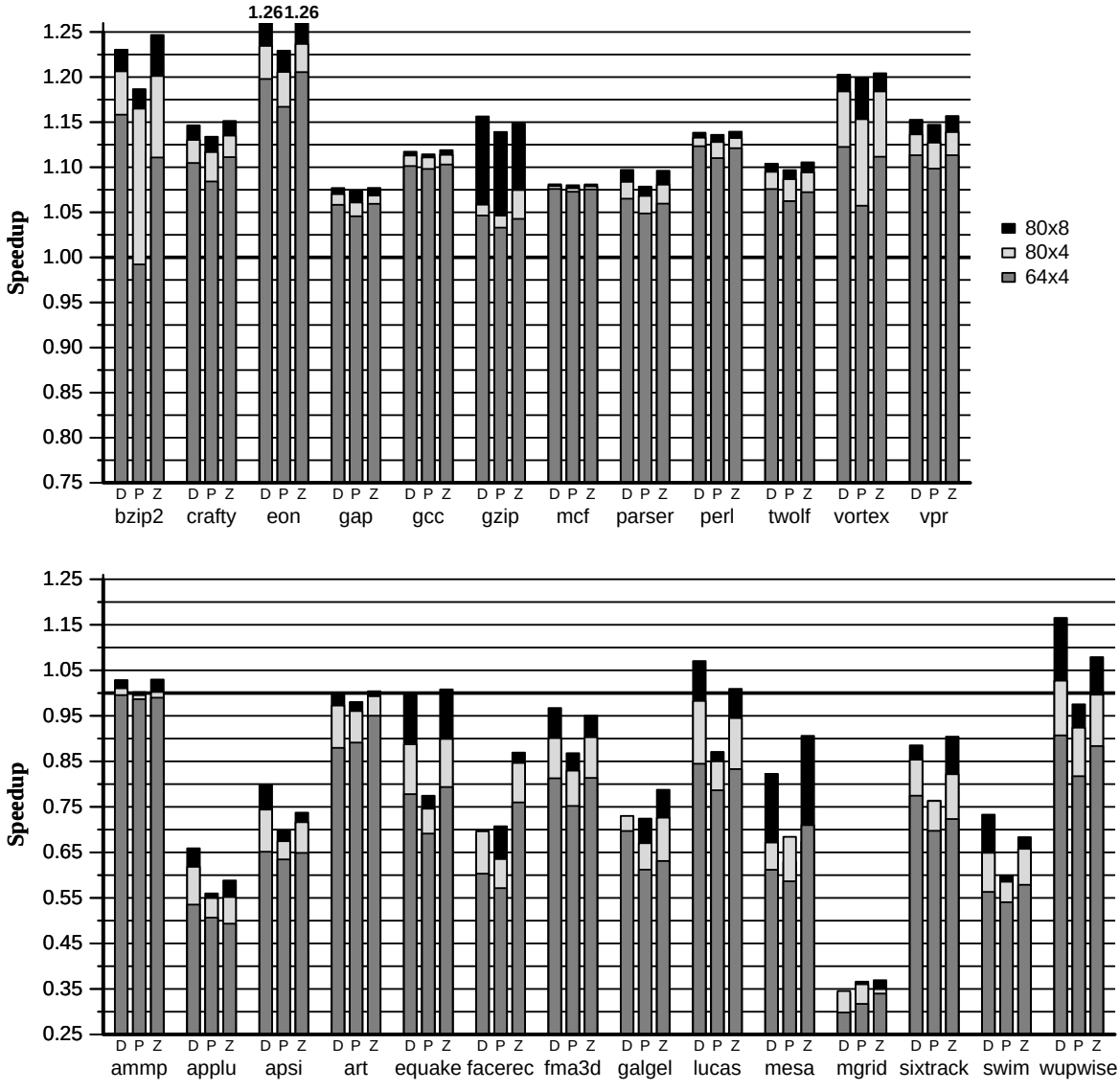


Figure 5.8. Hybrid replacement policies

performance of these two schemes. As in Figure 5.7, D indicates fewest-use replacement. P corresponds to a modification of FIFO in which values with the maximum degree of use are pinned in the cache; Z shows the performance of a modification of LRU in which entries with zero remaining uses are given priority. The latter scheme performs competitively with and, in some cases, better than the pure use-based scheme. The performance of the pinned-FIFO replacement policy represents a slight improvement over classical FIFO for those benchmarks where capacity is not at a premium (i.e., mostly the integer benchmarks), but does not perform as well as the other two policies. Comparing the performance of these hybrid policies with that of the original use-based replacement policy suggests that the main advantage of the original use-based replacement policy over LRU or FIFO is its preferential selection of dead victims rather than its ability to keep high-use values in the cache.

5.4.6 Miss breakdown

Figure 5.9 shows a breakdown of register cache misses for three register cache algorithms. The configurations are drawn from Figure 5.6 and Figure 5.7: the A configuration inserts all values and uses LRU replacement, the N configuration performs non-bypass filtering with LRU replacement, and the D configuration uses use-based insertion filtering and use-based replacement. In this figure, the cache organization is fixed at four-way set-associative with a capacity of 80 entries (i.e., the 80x4 organization indicated by the light gray bars in the previous figures), and the different components of each bar represent the portion of the overall miss rate due to filtering (dark gray), capacity evictions (light gray), and conflict evictions (black).

Filtering misses are easily counted, but the classification of the eviction misses as conflict or capacity misses is more involved. For each cache algorithm, the miss rate was also determined using a fully-associative cache of the same capacity (80 entries), which does not suffer from conflict misses. The capacity miss rate of the set-associative cache was assumed to equal to the non-filtering miss rate from the fully-associative cache.† The remaining portion of the overall miss rate was attributed to conflicts.

---

† The validity of this assumption depends on the filtering miss rate being the same in the set-associative and fully-associative caches for each given algorithm; the data indicate that this is a reasonable approximation (the median difference in the filtering miss rates is 3.3% of the set-associative cache's miss rate).

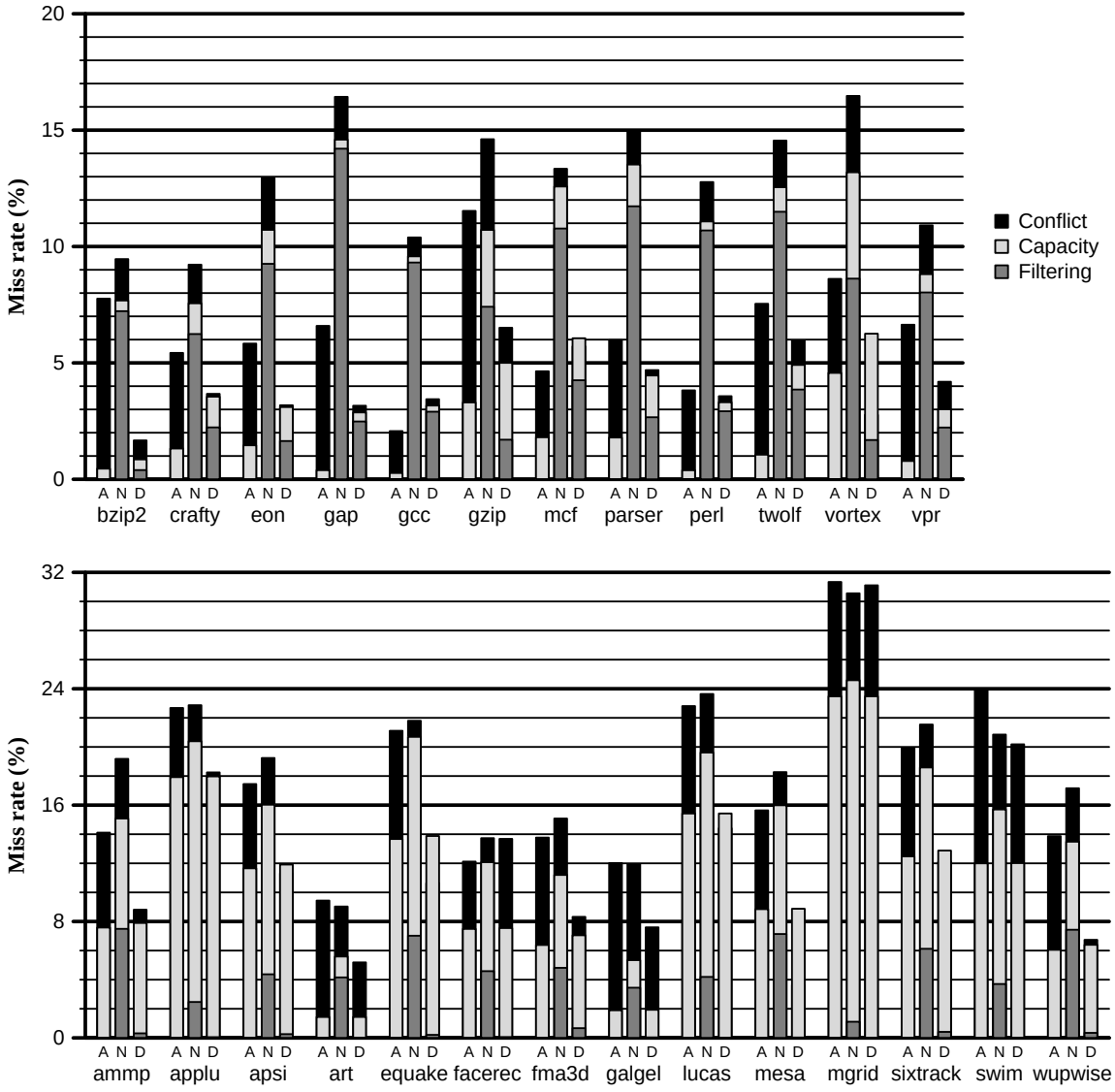


Figure 5.9. Register cache misses

The insertion policy in which all values are written to the cache (A bars) cannot cause filtering misses; in contrast, the filtering of any values by the non-bypass or use-based filtering policies must result in some such misses. The number of filtering misses introduced by use-based filtering is less than the decrease in eviction misses from the lower write bandwidth, leading to a lower overall miss rate and higher performance.† The opposite is true for the non-bypass scheme

† The data in Figure 5.9 do not strictly illustrate that the reduction in eviction misses is due to use-based filtering since the D bars also employ use-based replacement. However, the performance results of Figure 5.6, in which LRU replacement is used with different insertion policies, clearly show that the use-based insertion policy must be lowering the overall miss rate.

(N bars): here, the misses from filtering exceed the reduction in eviction misses due to that filtering, resulting in a performance loss.

Filtering misses represent a substantial portion of the miss rate of a use-based register cache, but only for the integer benchmarks. The register cache misses experienced by the floating-point benchmarks are dominated by capacity and, to a lesser extent, conflict misses. The greater capacity demands of the floating-point benchmarks result from a combination of fewer bypassed values (Section 5.3.1), a larger number of live values (Figure 5.1), and longer value lifetimes (Figure 2.4).

5.4.7 Sensitivity studies

Figure 5.10 isolates the effect of the register cache capacity by showing the performance of register caching using fully-associative register caches of different sizes. The use-based, non-bypass, and LRU curves correspond to the D, N, and A configurations of Figure 5.9, respectively. Floating-point and integer benchmarks are separated to illustrate the extreme difference in their behavior.

The use of a register cache as small as 32 entries improves the performance of the integer benchmarks over a three-cycle register file. For a 96-entry register cache, the performance

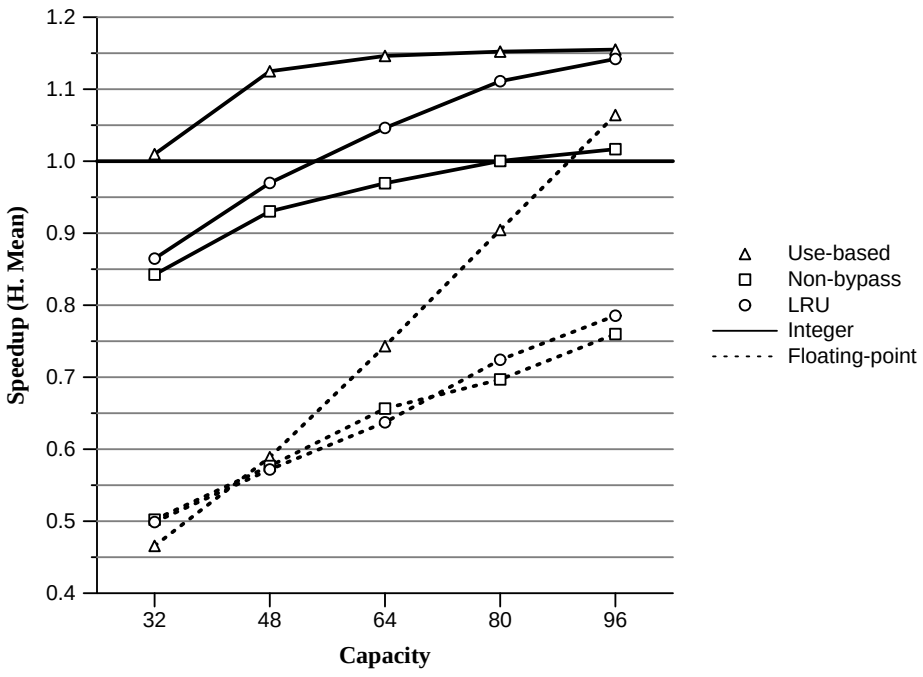


Figure 5.10. Cache capacity

improvement is about 15%, although the benefit reaches 12% with only 48 entries. Given the limited number of capacity misses exhibited by the integer benchmarks (Figure 5.9), the saturation of the performance with capacity is to be expected.

The floating-point benchmarks suffer terribly under register caching. With the use-based cache policies, 96 cache entries are required to exceed the performance of the three-cycle register file on the floating-point benchmarks; the other two caching algorithms still exhibit a slowdown of more than 20% at this capacity. In contrast with the integer benchmarks, the nearly linear increase in performance with capacity demonstrates the significant contribution of capacity misses to the behavior of the floating-point benchmarks.

The performance of register caching in a narrower pipeline is considered in Figure 5.11. The presentation is similar to that of Figure 5.6 except that the bars in each stack indicate the performance of 56×4 (dark gray), 64×4 (light gray), and 64×8 (black) register caches. The simulated parameters are identical to those presented in Table 5.1 except: (1) the pipeline is four-wide, (2) the machine can fetch from one cache line per cycle and a single taken branch terminates fetch for that cycle, (3) the reorder buffer, register file, and backing file have 256 entries, (4) the issue window has 128 entries, and (5) the core configuration matches the rich resource configuration from Table 4.4. The register cache capacities have also been reduced from the 64- and 80-entry sizes to maintain the same relative sizes versus the smaller physical register file.

The behavior of the register cache algorithms in a four-wide machine is similar to that in the eight-wide machine depicted in Figure 5.6 and Figure 5.7. The performance of use-based register caching relative to the multi-cycle register file baseline or to the other caching algorithms is reduced versus the wider machine. This phenomenon may be explained by noting that as the execution bandwidth is decreased, there are fewer opportunities for bypassing, reducing the potential of use-based filtering. Simultaneously, value lifetimes increase, leading to a greater window of vulnerability in which a poor decision can lead to a miss. In spite of the reduced benefit, use-based caching still delivers the best performance of the three register caching algorithms. Speedups are achieved for all of the integer benchmarks, even with a 56×4 register cache (using the use-based policies). Of the floating-point benchmarks, only ammp achieves a speedup under any policy for the range of cache organizations tested.

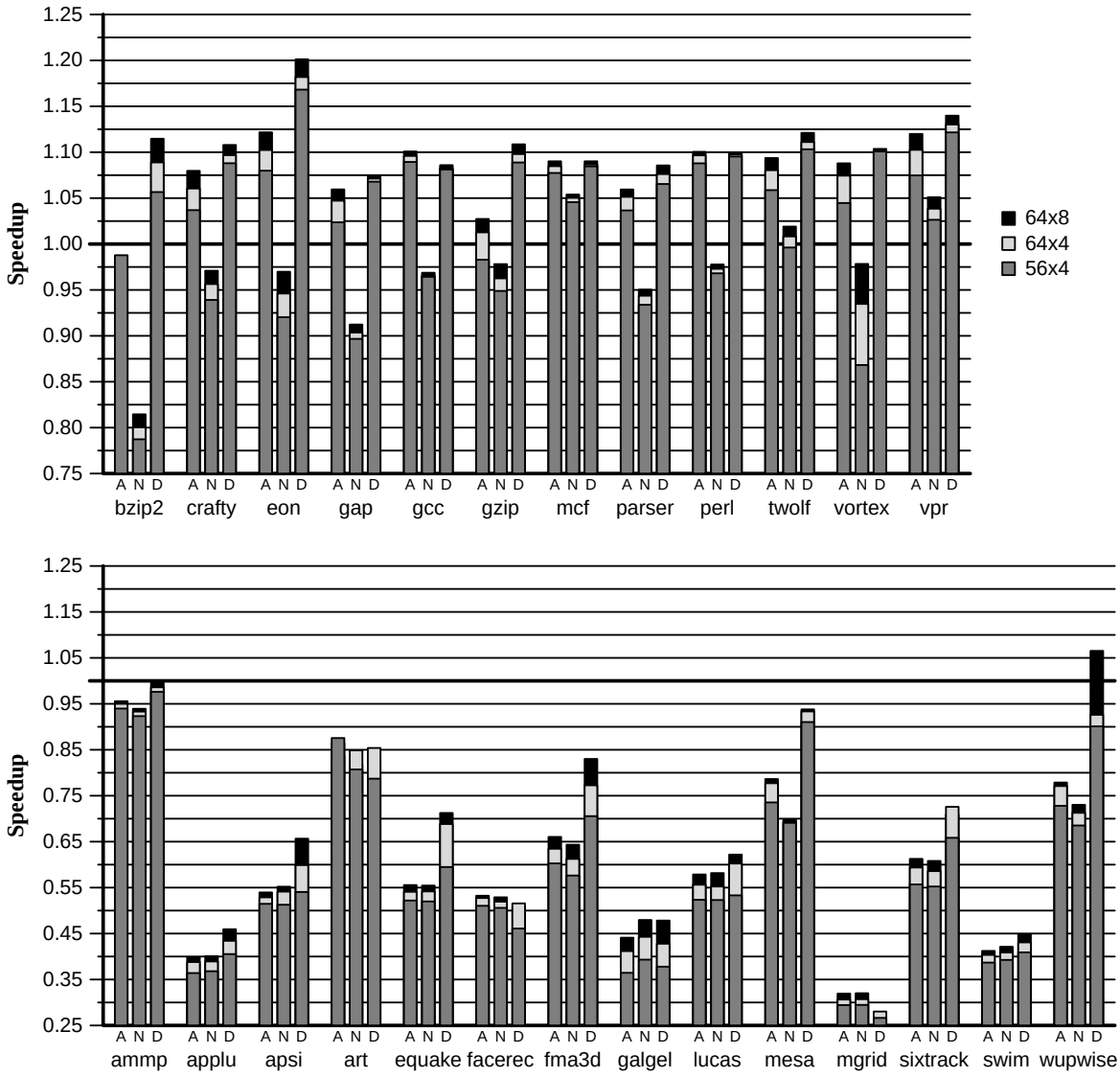


Figure 5.11. Register cache performance in a four-wide machine

5.5 Related Work

Due to the central role of the register file, a huge body of research has been aimed at optimizing this structure, particularly its access time. Many of these proposals include some form of banking or clustering to divide the register file's bandwidth (and sometimes capacity) requirements among several smaller structures. This work is largely orthogonal to the use of a register cache: many of the schemes are as applicable to a register cache as a register file. Here, the discussion is limited to register hierarchies and proposals that leverage the occurrence of empty and dead values within the register file (observed in Section 5.1).

A register cache is a particular implementation of a general class of multi-level register hierarchies. Like memory hierarchies, multi-level register hierarchies all provide for different classes of register storage in an attempt to reduce the average register read latency. The methods differ on such attributes as the structure of the register hierarchy, whether inclusion is enforced, and, most importantly, how values are managed within the hierarchy. Some of these schemes depend on explicit software assignment of values to different levels in the hierarchy [58, 81, 93]. Such schemes not only necessitate compiler support, but also require exposing the register file implementation to the ISA.

Previous examples of hardware register caches that operate essentially as described in Section 5.2 have been proposed. Yung and Wilhelm [92] first suggested a two-level register hierarchy in which a small, fast register cache that communicates directly with the execution core is accompanied by a backing store. They evaluated a fully-associative register cache managed using an LRU policy. Cruz et al. [24] invented the non-bypass heuristic to keep bypassed values from polluting the small register cache. Their cache was a fully-associative structure, but used pseudo-LRU replacement. They also proposed two prefetching schemes to bring results into the register cache before they are needed.

The distributed register algorithm by Borch et al. [11] employs multiple, fully-associative FIFO register caches in a clustered architecture. Some writes are avoided for values whose observable consumers: (1) can obtain the result from the forwarding buffer (bypass network), (2) can obtain the result from the register file prior to issue, or (3) will not execute on a different cluster.

Postiff et al. described a rather more complicated register caching scheme that amounts to a small, exclusive, direct-mapped register cache with a special index assignment policy [69]. Because their register cache is exclusive, a cache entry must not be reassigned until prior writer of that entry commits; at that time, the result value is written to the larger backing store and the cache entry may be reassigned.

Balasubramonian et al. introduced another exclusive register file hierarchy in which dead register values are moved from the physical register file (L1) to a backing structure (L2) with a hidden namespace [8]. Dead register values are moved when the number of L1 registers drops below

a predetermined threshold. Control mis-speculations and exceptions require registers from the L2 to be copied back into the L1.

Many other researchers have also taken aim at the empty and dead values occupying the register file. In some cases, software annotations are used to indicate that the register values are dead earlier than would be otherwise known [57, 58, 60], allowing them to be released early. Monreal et al. devised a scheme for discovering these early release conditions dynamically [64]. Some implementations simply rely upon the storage of the dead values in a checkpoint [3, 61] or elsewhere [56] if needed for recovery. González et al. [35, 63] and Wallace and Bagherzadeh [87] attack the empty physical registers instead, proposing to delay the allocation of a physical register to a result until after the result has been generated.

5.6 Summary

Register caching is a technique to address the difficulties in implementing large, low-latency register files and their associated bypass networks. Large register files are required to support an increasing number of in-flight instructions in deep pipelines with high execution bandwidth. As clock frequencies increase, however, the access latency of the register file extends across multiple pipeline stages. Bypass networks suffer similar scaling issues, preventing the increased register latency from being fully-hidden. A register cache leverages the fact that a large fraction of allocated registers are either empty or contain a value that has already been read by all of its consumers. By storing these values only, the register cache can maintain low latency, and full bypassing can be supported.

The successful operation of a register cache depends on the ability to keep it filled with this set of live values. Controlling the cache's contents is the province of its insertion policy, which decides what values will enter the cache, and its replacement policy, which decides what entry will be replaced during an insertion. Ideally, insertions would be restricted to live values and replacements would preferentially select values that had become dead since entering the cache. The number of remaining uses of a value, derived from comparing the actual uses with the information from a degree of use predictor, indicates whether or not it is live. This information can then be used by cache management policies to approximate the ideal behavior.

This chapter described two such policies for the management of the contents of a register cache. Use-based filtering exploits the fact that the bypass network performs a significant fraction of all value communication. Due to the structure of the pipeline, many of these bypasses can be observed and accounted for before the value must be written to the register cache. If the number of observable bypasses equals or exceeds the predicted degree of use, the value can avoid entering the cache and causing a replacement. When a replacement is necessary, use-based replacement attempts to select those values that are either dead (i.e., have no remaining uses) or will be least likely to lead to many future misses.

These policies substantially improve the performance of register caching over previous proposals for register cache capacities of interest. Versus a three-cycle 320-entry register file, a 64-entry, four-way set-associative use-based register cache yields a 9.3% speedup on the integer benchmarks. Applying the best previously-proposed policy to the same cache results in a speedup of only 2.7%. Performance on the floating-point benchmarks was significantly lower than when using a multi-cycle register file, but it was still superior to that offered by previous policies. The behavior of floating-point benchmarks was attributed to a much higher number of live values, which led to a large number of capacity misses.

Chapter 6

Conclusions

The complexity of current inter-instruction value communication mechanisms represents the most important barrier to the implementation of future high performance processors. This complexity arises because every value is treated exactly the same. Identical resources are used for the communication of each instruction's result, and the value communication structures support the most general possible communication behavior of each value. Register files assume the need for long-term storage, bypass networks assume high fan-out, and instruction windows assume that all waiting instructions could use every result. The consequences of these assumptions are reflected in the difficulty in scaling these structures to exploit more parallelism at ever higher frequencies.

Degree of use offers a method by which the degeneracy among values can be broken. It is a simple, intuitive indicator of the nature of a value's participation in inter-instruction communication. Some values undoubtedly need the general high-powered communication capabilities provided to every value now—but these are few in number. A far greater portion of values have modest requirements, being used once or twice within a short time of their generation. Degree of use information differentiates these behaviors, allowing for the dynamic selection among mechanisms adapted to the needs of specific kinds of values.

This dissertation has presented an in-depth exploration of the characteristics, prediction, and application of degree of use information. Section 6.1 summarizes these contributions. Section 6.2 describes other potential optimizations enabled by degree of use knowledge, and

Section 6.3 concludes with a discussion of the factors that determine the benefit of use-based communication optimizations in general.

6.1 Contributions and Key Results

The major contributions of this work were: a characterization of the properties of degree of use and its relationship to inter-instruction communication (Chapter 2), the description of static and dynamic methods for high-accuracy degree of use prediction (Chapter 3), a characterization of and mechanism for exploiting zero-use values (Chapter 4), and the demonstration of a superior method of register caching based on the exploitation of degree of use information (Chapter 5). The key results of each of these contributions is summarized below.

6.1.1 Degree of use characterization

The initial definition and exploration of degree of use was presented by Franklin and Sohi [32]. The characterization presented in this work both confirms and significantly expands upon their results. The dominance of simple communication patterns in programs was demonstrated to hold across languages, compilers, and individual programs. Single-use values comprise the majority of all values generated during a program's execution. Values with greater numbers of uses occur progressively less frequently. While high-use values are less frequent, they supply proportionally more consumers, indicating that efficient means for the wide distribution of values are needed.

New insight into the degree of use properties of values was obtained by correlating their degree of use to individual instructions within the program. Certain kinds of values, identified by the architectural registers to which they were bound or the type of instructions generating them, exhibit behaviors quite different than the overall average. Addresses, for example, have a significantly higher average degree of use than other values. The examination of degree of use on a per-instruction basis also revealed the existence of the locality necessary to achieve accurate degree of use prediction.

Mathematical models of degree of use properties were also considered. Previous researchers had proposed that the frequency distribution of values with different degrees of use was fit by a power-law model [28]. This work extended the prior model to account for zero-use values, simultaneously providing a useful probability distribution function for the occurrence of values with

different degrees of use. A typical application of such a model was demonstrated during the comparison of different register cache replacement algorithms (Section 5.3.2).

6.1.2 Degree of use prediction

The demonstration of accurate degree of use prediction is the cornerstone of this work, enabling the implementation of the speculative inter-instruction communication optimizations presented in the latter chapters. How each instruction fits within the overall communication structure of the program is fixed when the program is compiled. Therefore, the range of behaviors of values arising from a particular instruction is predetermined and can be discovered by static dataflow analysis. The definition of the degree of use dataflow problem for performing this analysis represents an important contribution of this dissertation with an immediate application to static degree of use prediction.

Static degree of use prediction is the compile-time assignment of a single degree of use to some or all of the value-generating instructions within a program. Many instructions have a unique statically-determinable degree of use, making such an assignment straightforward. However, where dataflow analysis indicates multiple possible outcomes, a decision must be made on which single degree of use to select, if any. A combination of heuristics and profiling information may be used to guide such a decision. One such method for merging profile data with static analysis was presented and demonstrated to yield good results in terms of static prediction accuracy and coverage.

Dynamic degree of use prediction offers an alternative means of obtaining degree of use information. The use of dynamic prediction: (1) removes the requirement that analysis be performed for each program of interest, (2) avoids the need to communicate the information from the static analysis tool(s) to the runtime system, and (3) eliminates the constraint of a single degree of use per static instruction. The success of dynamic degree of use prediction is predicated on the existence of per-instruction locality in degree of use behavior, which was amply demonstrated in Chapter 2. Three dynamic prediction mechanisms were described representing a spectrum of complexity and performance possibilities. The best-performing mechanism relies upon future control-flow information, a novel contribution of this work with likely applications beyond degree of use prediction.

6.1.3 Useless instruction elimination

Degree of use prediction was first applied to the exploitation of useless instructions. Useless instructions refer to dynamic instructions that generate zero-use values, which account for more than 10% of the dynamic instruction count in some optimized benchmarks. This phenomenon was investigated further, and the dominant cause of useless instructions was found to be the introduction of partially-dead instructions during compiler optimization.

Consideration of the resources wasted in handling useless instructions motivated the development of useless instruction elimination, which is a mechanism whereby useless instructions, identified through degree of use prediction, can be retired without executing. Useless instruction elimination is representative of a general type of use-based optimization: namely, the special handling of a certain class instructions based on a predicted property of their result values. The mechanism ensures that a candidate useless instruction can be safely removed by executing speculatively until: (1) the prediction is verified by the overwrite of the value, (2) a use of the value is encountered, or (3) further speculation is blocked by resource limitations. In the first case, the candidate is retired having avoided execution; either of the latter two cases result in the delayed execution of the candidate.

The performance benefit of useless instruction elimination is generally small and highly-dependent on contention for issue and execution resources in the microarchitecture. Performance losses are limited to a fraction of a percent where resources are abundant. In a resource-poor microarchitecture, average speedups of a few percent were realized. Another benefit is the reduction in resource utilization. The frequency of L1 data cache accesses, register file writes and reads, issue bandwidth, and instruction executions are all reduced by about the same percentage as the incidence of useless instructions.

6.1.4 Use-based register caching

Use-based register caching represents an application of degree of use information in optimizing actual value communication. The large and slow physical register file is replaced by a small cache, which ideally contains only those values that will be used (i.e., live values). The contribution of this work over previous register caching proposals is in the nature of the cache insertion and replacement policies.

Both the insertion and replacement policies rely on determining the usefulness of a particular value. Degree of use knowledge makes this possible by providing the total number of uses expected prior to the generation of the value; once generated, the expected number of uses can be adjusted as actual uses occur, leaving an exact count of the remaining uses. The register cache insertion policy uses this information to explicitly account for the bypass network as an alternative means of value communication: values that reach all of their consumers via the bypass network are easily detected (they have zero remaining uses upon reaching the cache) and are kept from polluting the cache. Provided a live value remains in the cache long enough, it will eventually reach all of its consumers. At this point the value is no longer needed in the cache and should yield to incoming live values. Use-based replacement selects victims based on the number of remaining uses, preferring those with fewer uses left. Modulo the accuracy of the remaining use counts, this policy will always replace dead values over live ones. In this application, use-based register caching represents a vast improvement over prior proposals, offering better performance for all benchmarks and cache sizes of interest.

6.2 Additional Applications of Degree of Use Knowledge

This section suggests other potential applications of the information provided by degree of use information. Possible improvements to useless instruction elimination and use-based register caching were addressed in their respective chapters and are not reiterated here. These represent what I believe to be promising avenues for investigation, but their potential has not been experimentally verified. In many cases, the optimizations offer an alternative to or an improvement of previously-proposed mechanisms that were motivated by the properties demonstrated in Chapter 2, but did not have the benefit of explicit per-value use information.

6.2.1 Early register reclamation

Degree of use prediction offers a mechanism for identifying the last use of a value. As was done in use-based register caching, predictions initialize per-value counts of expected uses, which are subsequently adjusted as those uses occur. This information can enable the speculative early reclamation of physical registers with appropriate attention to mis-speculation recovery. Previous proposals for early register release require a delay until the observation of the instruction over-

writing the corresponding architectural register to ensure that no further uses will occur [3, 8, 61, 64]. When using the method based on the degree of use prediction, however, this delay is not necessary as the occurrence of the last use is made explicit. This advantage may allow for more aggressive recycling of physical registers.

6.2.2 Registerless communication

Given the abundance of degree of use one values, mechanisms exploiting their existence should be widely applicable. Regarding the actual communication of these values, it is obvious that use of the register file results in unnecessary overhead. The register communication model implicitly (but incorrectly) suggests that a value bound to a register will be used multiple times. Values with a predicted degree of use of one need not even use a register.

With proper attention to the scheduling of the producer and consumer operations, the communication of such values can occur entirely through the bypass network. Use-based register caching represents a small step towards this end, but all values still consume storage in the backing file. Given efficient mechanisms for value for mis-speculation recovery (addressed in Section 6.3) and good heuristics for choosing which values will receive registers, it should be possible to limit performance loss from mis-speculation to a reasonable level. The benefit would be obtained through a combination of the reductions in the number of registers required and the number of register file write ports, which would significantly reduce the size of the register file, allowing it to be faster and/or lower power.

6.2.3 Collapsing dependent operations

The knowledge that the value communicated between two instructions is private (i.e., has a degree of use of one) can also be exploited to dynamically collapse dependent operations. Given simple enough operations (e.g., dependent logical operations), it is quite possible to complete both operations in a single cycle. The resulting reduction in the dataflow height could result in increased performance. Interlock collapsing ALUs have been proposed as a means of executing two dependent operations together [59, 68, 73], but the application of this technique is limited by the need to generate and store the intermediate value or statically ensure that the dependent operation is the only consumer. A degree of use predictor can increase the applicability of this technique by identifying such instances dynamically.

6.2.4 Direct consumer scheduling

The scheduling of dependent operations themselves could also be simplified with knowledge of degree of use. Instructions with a predicted degree of use of one can be allocated dedicated reservation stations that are directly addressable by the completing parent instruction. The dependent instruction can be steered to this reservation station by information available at the rename stage. Upon completion of the parent instruction, the wakeup operation would not require a tag broadcast across a large associative instruction window. Instead, the proper dependent operation could be woken up directly.

6.2.5 Widely-used values

Figure 2.3 shows that the small number of values with a high degree of use contribute significantly to the total number of values read as instruction inputs. This phenomenon suggests that these values should migrate or be allocated to structures that can deliver them to their consumers earlier or with less overhead than a register file.

One possibility is to maintain a very small storage structure associated with each ALU to supply these values. With a very small capacity, such a structure could significantly reduce the bandwidth demands on the power-hungry register file. This is similar to the local register files proposed by Franklin and Sohi [32] but at a finer granularity. Rather than using a per-cluster local register file to maintain values belonging to that cluster, a high-use value cache would maintain only values having more than a certain number of uses.

Another potential way to exploit high-use values is to copy or migrate them towards the front end. Values with many uses are likely to live long enough to reach the front end while they are still actively mapped. In these cases, it is possible to access them directly using the architectural register identifier instead of using a physical register tag obtained through renaming. This is similar to physical register inlining [56], but it does not depend on the size of the value. The availability of some values early in the pipeline may enable additional optimizations such as early execution.

6.3 Costs and Benefits of Use-Based Communication Optimizations

The various value communication optimizations enabled by degree of use prediction attack the *complexity* of value communication. Performance improvements are achievable only to the extent that the complexity affects performance. A pipelined, superscalar processor capable of supporting single-cycle (or fully-bypassed) register operations for all simultaneously-executing instructions places an upper bound on performance. Modulo pipeline bubbles and hazards, such a machine will execute from the available instruction window as fast as data dependences (and execution resource constraints) allow. The problem is not the peak performance of this machine, but the inability to implement it. Optimized communication mechanisms may enable higher clock frequencies or the realization of a feasible design that approaches this limit.

Use-based optimizations enable the use of less complex structures, but they are inherently speculative. In order to achieve performance as close as possible to that of an ideal machine, then, two factors must be minimized: (1) the number of mis-speculations, and (2) the cost of mis-speculation recovery. The predictor designs offered in Chapter 3 deliver very high accuracy with respect to the degree of use itself. Any optimizations using this information to engage in further speculation must be careful to maintain similar accuracies. Minimizing the cost of mis-speculation recovery is more complex.

The obvious cost of mis-speculation recovery is incurred upon an actual mis-speculation. The cost of this recovery will be an important determinant of the success of a speculative optimization. In the case of speculative communication optimizations, recovery entails obtaining a specific value that was not communicated correctly (and potentially re-executing instructions that received an incorrect value). There are basically two approaches: the value may be regenerated (e.g., aborting a useless instruction elimination) or the value may be obtained elsewhere (e.g., the backing file for use-based register caching). As was the case for the two optimizations presented in this dissertation, the choice of recovery mechanism is dependent on the semantics of the particular optimization.

A more subtle cost of mis-speculation recovery is the overhead required to maintain the *ability* to recover from mis-speculations. This cost was particularly egregious for useless instruction elimination, where retirement had to be stalled to ensure that any unverified eliminated instruction could be re-executed. In the case of use-based register caching, this cost manifested as the need to

write every value to the backing register file. Microarchitectures designed from the beginning to support aggressive speculation (e.g., those employing checkpointing [3, 61]) are likely to be the most appropriate substrate for communication optimizations.

Finally, it should be recognized that although use-based communication optimization seeks to attack complexity, it can also be a source of complexity. The simplification of a complex communication structure must be balanced against the inclusion of special-case communication structures with limited marginal utility. Again, it is likely to be the support for mis-speculation recovery rather than the particular communication mechanism itself that will drive this trade-off.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000. pp. 248-59.

[2] P. Ahuja, D. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995. pp. 36-45.

[3] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint processing and recovery: towards scalable large instruction window processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, December 2003. pp. 423-34.

[4] *Alpha 21264/EV67 Hardware Reference Manual*, Compaq Computer Corporation, March, 2002.

[5] *Alpha Architecture Handbook, 4th Ed.*, Compaq Computer Corporation, January 2002.

[6] *Assembly Language Programmer's Guide*, Digital Equipment Corporation, March 1996.

[7] G. Ammons and J. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1998. pp. 72-84.

[8] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December, 2001. pp. 237-48.

[9] R. Bodik and R. Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1997. pp. 159-70.

[10] M. Bohr. Interconnect scalaing: the real limiter to high performance ULSI. In *Proceedings of the IEEE International Electron Devices Meeting*, December 1995. pp. 241-4.

[11] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *Proceedings of the 8th Annual International Symposium on High-Performance Computer Architecture*, February 2002. pp. 270-81.

[12] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.

[13] J. Burns, and J.-L. Gaudiot. Quantifying the SMT layout overhead—does SMT pull its weight? In *Proceedings of the 6th Annual International Symposium on High-Performance Computer Architecture*, January 2000. pp. 109-20.

[14] J. Butts and G. Sohi. Characterizing and predicting value degree of use. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, November 2002. pp. 15-26.

[15] J. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Proceedings of the 10th International Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2002. pp. 199-210.

[16] J. Butts and G. Sohi. A static power model for architects. In *Proceedings of the 33rd Annual International Sympoisum on Microarchitecture*, December 2000. pp. 191-201.

[17] J. Butts and G. Sohi. Use-based register caching with decoupled indexing. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004. pp. 302-13.

[18] H. Cain, K. Lepak, B. Schwartz, and M. Lipasti. Precise and accurate processor simulation. In *Proceedings of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads*, February 2002. pp. 13-22.

[19] H. Cain and M. Lipasti. Memory ordering: a value-based approach. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004. pp. 90-101.

[20] P. Chang, N. Warter, S. Mahlke, W. Chen, and W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Transactions on Computers*, 44(3), March 1995. pp. 481-94.

[21] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. Ju, and J. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, October 2003. pp. 25-36.

[22] W. Chen, S. Mahlke, N. Warter, S. Anik, and W. Hwu. Profile-assisted instruction scheduling. *International Journal for Parallel Programming*, 22(2), April 1994. pp. 151-81.

[23] K. Cooper, M. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the IEEE 1992 International Conference on Computer Languages*, April 1992. pp. 96-105.

[24] J.-L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000. pp. 316-25.

[25] A. Dhodapkar and J. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002. pp. 233-44.

[26] K. Driesen and U. Hoelzle. The cascaded predictor: economical and adaptive branch target prediction. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998. pp. 249-58.

[27] A. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998. pp. 69-77.

[28] L. Eeckhout and K. Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, September 2001. pp. 25-34.

[29] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May, 2002. pp. 37-46.

[30] M. Evers, S. Patel, R. Chappell, and Y. Patt. An analysis of correlation and predictability: what makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998. pp. 52-61.

[31] A. Falcón, J. Stark, A. Ramirez, K. Lai, and M. Valero. Prophet/critic hybrid branch prediction. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004. pp. 250-61.

[32] M. Franklin and G. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992. pp. 236-45.

[33] J. Fisher and S. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the 5th International Symposium on Architectural Support for Programming Languages and Operating Systems*, September 1992. pp. 85-95.

[34] J. Fu, J. Patel, and B. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992. pp. 102-110.

[35] A. González, J. González, and M. Valero. Virtual-physical registers. In *Proceedings of the 4th Annual International Symposium on High-Performance Computer Architecture*, February 1998. pp. 175-84.

[36] R. Gupta, D. Berson, and J. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997. pp. 358-68.

[37] A. Hartstein and T. Puzak. The optimum pipeline depth for a microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002. pp. 7-13.

[38] J. Hennessy and D. Patterson. *Computer Architecture, a Quantitative Approach, 2nd Ed.* Morgan Kaufmann, 1996.

[39] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. In *Intel Technology Journal, Q1*, Intel Corporation, 2001.

[40] G. Hinton, M. Upton, D. Sager, D. Boggs, D. Carmean, P. Roussel, T. Chappell, T. Fletcher, M. Milshtein, M. Sprague, S. Samaan, and R. Murray. A 0.18-μm CMOS IA-32 processor with a 4-GHz integer execution unit. In *IEEE Journal of Solid-State Circuits*, 36(11), November 2001. pp. 1617-27.

[41] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002. pp. 14-24.

[42] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002. pp. 209-20.

[43] Z. Hu and M. Martonosi. Reducing register file power consumption by exploiting value lifetime characteristics. Presented at the Workshop on Complexity Effective Designs (held in conjunction with the 27th Annual International Symposium on Computer Architecture), June 2000.

[44] *IA-32 Intel Architecture Software Developer's Manual*, volume 2, Intel Corporation, 2001.

[45] *International Technology Roadmap for Semiconductors, Executive Summary*. Semiconductor Industry Association, 2003.

[46] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996. pp. 142-52.

[47] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990. pp. 364-73.

[48] R. Kessler. The Alpha 21264 microprocessor. In *IEEE Micro*, 19(2), April 1999. pp. 24-36.

[49] G. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, October 1973. pp. 194-206.

[50] H. Kim and J. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002. pp. 71-81.

[51] A. Klaiber. *The technology behind Crusoe processors*. Transmeta Corporation White Paper, January 2000.

[52] A. KleinOsowski and D. Lilja. MinneSPEC: a new SPEC benchmark workload for simulation-based computer architecture research. In *Computer Architecture Letters*, June 2002.

[53] J. Knoop, O. Ruthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994. pp. 147-58.

[54] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, May 1981. pp. 81-7.

[55] K. Lepak and M. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000. pp. 182-91.

[56] M. Lipasti, B. Mestan, and E. Gunadi. Physical register inlining. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004. pp. 325-35.

[57] J. Lo, S. Parekh, S. Eggers, H. Levy, and D. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. In *IEEE Transactions on Parallel and Distributed Systems*, 10(9), September 1999. pp. 922-33.

[58] L. Lozano C. and G. Gao. Exploiting short-lived variables in superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995. pp. 292-302.

[59] N. Malik, R. Eickemeyer, and S. Vassiliadis. Interlock collapsing ALU for increased instruction level parallelism. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992. pp. 149-57.

[60] M. Martin, A. Roth, and C. Fischer. Exploiting dead value information. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997. pp. 125-35.

[61] J. Martínez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, November 2002. pp. 3-14.

[62] E. Mehofer and B. Scholz. Probabilistic data flow system with two-edge profiling. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, July 2000. pp. 65-72.

[63] T. Monreal, A. González, M. Valero, J. González, and V. Viñals. Delaying physical register allocation through virtual-physical registers. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, November, 1999. pp. 186-92.

[64] T. Monreal, V. Viñals, A. González, and M. Valero. Hardware schemes for early register release. In *Proceedings of the International Conference on Parallel Processing*, August 2002. pp. 5-13.

[65] T. Mudge. Power: a first-class architectural design constraint. In *IEEE Computer*, 34(4), April 2001. pp. 52-8.

[66] S. Palacharla, N. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997. pp. 206-18.

[67] D. Papworth. Tuning the Pentium-Pro microarchitecture. In *IEEE Micro*, 16(2), April 1996. pp. 8-15.

[68] J. Philips and S. Vassiliadis. High performance 3-1 interlock collapsing ALUs. In *IEEE Transactions on Computers*, 43(3), March 1994. pp. 257-68.

[69] M. Postiff, D. Greene, S. Raasch, and T. Mudge. Integrating superscalar processor components to implement register caching. In *Proceedings of the 2001 International Conference on Supercomputing*, June 2001. pp. 348-57.

[70] G. Ramalingam. Data flow frequency analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1996. pp. 267-77.

[71] E. Rotenberg. Exploiting large ineffectual instruction sequences. Technical Report, North Carolina State University, November 1999.

[72] A. Roth, A. Mendelson, and R. Ronen. Dynamic techniques for load and load-use scheduling. In *Proceedings of the IEEE*, 89(11), November 2001. pp. 1621-37.

[73] Y. Sazeides, S. Vassiliadis, and J. Smith. The performance potential of data dependence speculation and collapsing. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996. pp. 238-47.

[74] M. Schlansker and B. Rau. EPIC: explicitly parallel instruction computing. In *IEEE Computer*, 33(2), February 2000. pp. 37-45.

[75] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, ed., *Program Flow Analysis: Theory and Applications*, Prentice-Hall, 1981. pp. 189-234.

[76] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998. pp. 259-71.

[77] J. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, May 1981. pp. 135-48.

[78] G. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995. pp. 414-25.

[79] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002. pp. 25-34.

[80] Standard Performance Evaluation Corporation. `http://www.spec.org`.

[81] J. Swensen and Y. Patt. Hierarchical registers for scientific computers. In *Proceedings of the 1988 International Conference on Supercomputing*, July 1988. pp. 346-53.

[82] R. Tarjan. Depth first search and linear search algorithms. In *SIAM Journal of Computing*, 1(2), June 1972. pp. 146-60.

[83] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, 11(1), January 1967. pp. 25-33.

[84] L. Torvalds et al. `arch/alpha/entry.S`. In *Linux kernel source code, version 2.6.0*, December, 2003.

[85] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, May 1995. pp. 392-403.

[86] D. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1991. pp. 59-70.

[87] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, October 1996. pp. 179-84.

[88] C. Webb. Subroutine call/return stack. In *IBM Technical Disclosure Bulletin*, 30(11), April 1988.

[89] K. C. Yeager. The MIPS R10000 superscalar microprocessor. In *IEEE Micro*, 16(2), April 1996. pp. 28-41.

[90] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999. pp. 42-53.

[91] A. Yoaz, R. Ronen, R. Chappell, and Y. Almog. Silence is golden? Presented at the *7th Annual International Symposium on High-Performance Computer Architecture*, January 2001.

[92] R. Yung and N. Wilhelm. Caching processor general registers. In *Proceedings of the International Conference on Computer Design*, October 1995. pp. 307-12.

[93] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-level hierarchical register file organization for VLIW processors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, December 2000. pp. 137-46.

[94] C. Zilles and G. Sohi. Understanding the backward slices of performance degrading instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000. pp. 172-81.

Appendix

Methodology

Details on the experimental methodology are presented in this appendix.

A.1 Benchmarks

Throughout this document, the experimental evaluations make use of the SPEC CPU 2000 benchmark suite [80]. The SPEC benchmark suite consists of 26 programs divided into an integer benchmark set and a floating-point benchmark set based on the dominant type of computation performed. The integer benchmark set consists of 12 programs, 11 of which are written in C and one of which (`eon`) is written in C++. The 14 programs in the floating-point benchmark suite are written in one of three languages: Fortran-77 (6 programs), Fortran-90 (4 programs), or C (4 programs).

A.1.1 Input data

The train inputs provided by SPEC were used in all experiments except for the gathering of the profile data in Section 3.3.3. The number of instructions executed to process the reference inputs is prohibitive given the slowdown of execution-driven simulation. The train inputs are significantly smaller, but still execute on the order of 40 billion instructions per benchmark when run to completion. While reduced reference inputs such as those made available by the University of Minnesota [52] are an alternative, reduced inputs were not available for all of the benchmarks at

the time of this writing. Also, given the bundling of the standard input sets with the SPEC distribution, availability and stability of the training inputs is assured.

A.1.2 Multiple-input benchmarks

Three of the integer benchmarks (`eon`, `perl`, and `vpr`) specify multiple, independent program runs as part of the training input. For these benchmarks, aggregate data, representing the consecutive execution of the required runs, is presented. Where simulation time constraints required partial execution of a benchmark (see Section A.4.2), some of the runs may not be represented if the instruction limit occurred prior to the beginning of that run. Therefore, the order of the multiple inputs is important. The order of the inputs used for `eon` was cook, rushmeier, kajiya; for `perl`, diffmail, perfect, scrabbl; for `vpr`, placement (`-place_only`), routing (`-route_only`). Due to limitations of the simulator infrastructure, all microarchitectural state (e.g., cache and branch predictor contents) is re-initialized prior to each run.

A.1.3 `perl` test input

The evaluation of static prediction in Section 3.3.3 included profiling data gathered on the test input set, which is also distributed by SPEC. Except for `perl`, each benchmark's test run requires only a single input and was executed to completion. `perl`'s test input is problematic for the simulation environment because it forks new processes. The input consists of a top-level script `test.pl` that forks the perl interpreter sequentially on each of 59 test scripts (which themselves execute multiple independent tests). Several of these scripts also attempt to fork additional copies of perl or run a system command as part of their test routines.

This process was modified for the simulation environment by eliminating the wrapper `test.pl` script and simply running the simulator consecutively on each of the scripts, similar to the handling of multiple input files described in Section A.1.2. Any single test that required forking an external process was not executed. The following individual tests were eliminated:

- Tests 12 and 13 in `lex.t`
- Test 11 in `split.t`
- Test 3 in `goto.t`
- All tests in `script.t` (i.e., this script was not run at all)

A.2 Benchmark Compilation

How the individual benchmarks are compiled can have a significant impact on the final behavior of the benchmark (e.g., see Figure 4.3). This section describes each of the compilers used in the generation of benchmarks for this work as well as the flags used for generating benchmarks at different optimization levels.

A.2.1 Compilers

The compilers are divided into two suites such that between the two suites, each benchmark is compiled with a different compiler. The suites consist of *vendor compilers*, originally written specifically to support the Alpha architecture, and retargettable, third-party compilers, which support many different architectures. As such, the vendor compilers offer better optimization and yield higher-performing benchmarks. Therefore, binaries generated by the vendor compilers are used in all experiments where the effect of the compiler is not specifically being investigated.

Table A.1 lists the compilers comprising each suite. The version information for each compiler comes directly from the compiler itself when executed with `-V` or `--version` as appropriate. Note that the Fortran-77 and Fortran-90 benchmarks were compiled with the same compiler in each suite.

The flags listed for each compiler were used for all runs of that compiler, independently of additional flags affecting optimization. Most important is the flag that specifies the binary is to be statically-linked (`-non_shared`/`-static`/`-Bstatic`), which is required by the simulator

Table A.1: Compiler Suites

Suite	Lang.	Compiler	Flags
Vendor	C	Compaq C V6.4-214 (dtk) on Digital UNIX V4.0F (Rev. 1229) Compiler Driver V6.4-014 (dtk) cc Driver	-std -arch ev6 -non_shared -g3
	C++	DIGITAL C++ V6.1-027 on DIGITAL UNIX V4.0 (Rev. 1229)	-arch ev6 -non_shared -g3
	Fortran	DIGITAL Fortran 90 V5.2-705	-arch ev6 -non_shared -g3
Third-party	C	GNU gcc 3.3.3	-mcpu=ev6 -static -g3
	C++	GNU g++ 3.3.3	-mcpu=ev6 -static -g3
	Fortran	NAGWare Fortran 95 compiler Release 4.2(513)	-Bstatic -target=ev56

(see Section A.4.4). A target implementation is specified for each compiler (`-arch`/`-mcpu`/`-target`), which enables the generation of certain implementation-dependent instructions. In each case, the latest Alpha implementation supported by each compiler was selected in order to use the more efficient instructions that these implementations offer. The remaining flags specify the inclusion of debugging information provided it does not interfere with optimization (`-g3`) and the legality of certain source language constructs appearing in the benchmarks (`-std`).

A.2.2 Optimization levels

Three different optimization levels were used in the studies on useless instructions presented in Section 4.1.2. On all compilers, the unoptimized benchmarks were generated with the `-O0` flag, and the lightly-optimized benchmarks with `-O1`. The tuned configuration uses per-benchmark flags that yielded the highest performance on a simulated eight-wide, deeply-pipelined configuration (see Section A.5). The options used for the tuned configuration appear in Table A.2.

In many cases, the best performance was obtained by allowing deviation from IEEE standard floating-point behavior (e.g., with such options as `-fast`, `-ffast-math`, and `-ieee=nonstd`). Changes in floating-point behavior frequently result in slight differences between the outputs of a benchmark when compiled with different options. SPEC recognizes this possibility and provides per-benchmark tolerances (absolute and/or relative); any benchmark run must match the baseline output they provide within these tolerances. This restriction was observed in the selection of these optimization flags: optimizations that led to differences with the SPEC-provided benchmark outputs (when executed on a native Alpha machine) were not used.

A bug in the gcc compiler (not the benchmark) required one additional command-line option to be specified for most of the benchmarks using the `-ffast-math` option. The implementation of the `-ffast-math` option uses a special library, and the bug results in the library being specified improperly to the linker, resulting in a compile-time error. A modified gcc spec file (which specifies how the different phases of the compilation are run) was generated with the correct library ordering; the additional command-line option `-specs=specs.math` tells gcc to use a modified spec file `specs.math`. This option does not affect the optimization of the benchmark and is therefore not listed in the tables.

Table A.2: Tuned-Benchmark Compilation Options

Benchmark	Language	Vendor compiler	Third-party compiler
ammp	C	-O4 -fast -om	-O3 -funroll-loops -ftracer -ffast-math
art	C	-fast -inline speed	-O2 -funroll-loops -ftracer
bzip2	C	-O4 -fast	-O2 -funroll-loops -ftracer -ffast-math
crafty	C	-fast -om	-O3 -ftracer -ffast-math
eon	C++	-O2 -fp_reorder -inline speed -assume trusted_short_alignment	-O2 -funroll-loops -ftracer
equake	C	-O4 -inline speed -om	-O2 -ffast-math
gap	C	-fast -inline speed -om	-O3 -ffast-math
gcc	C	-fast -om	-O2 -ffast-math
gzip	C	-O3 -inline speed	-O2 -funroll-loops -ftracer -ffast-math
mcf	C	-O4 -fast -inline speed	-O3 -funroll-loops -ffast-math
mesa	C	-fast	-O2 -ftracer
parser	C	-fast -inline speed -om	-O3 -funroll-loops -ffast-math
perl	C	-O4 -fast -om	-O2 -ftracer
twolf	C	-fast	-O3 -funroll-loops -ftracer
vortex	C	-O4 -fast -om	-O3 -ftracer
vpr	C	-O4 -fast	-O3 -funroll-loops
applu	F77	-O5 -fast	-O4 -Ounsafe
apsi	F77	-O4 -fast	-O3 -ieee=nonstd
facerec	F90	-O4 -fast -om	-O3 -Oassumed -ieee=nonstd
fma3d	F90	-O4 -transform_loops	-O3 -Oassumed -ieee=nonstd
galgel	F90	-O4 -fast	-O3 -Ounsafe
lucas	F90	-fast -pipeline	-O4 -ieee=nonstd
mgrid	F77	-O4 -transform_loops -om	-O3 -Ounsafe
sixtrack	F77	-O4 -pipeline -om	-O3 -ieee=nonstd
swim	F77	-O5 -fast	-O4 -ieee=nonstd
wupwise	F77	-O4 -om	-O4 -ieee=nonstd

A.3 Binary Dataflow Analyzer

Two experiments required the results of the dataflow analysis described in Section 3.3.1. Besides the generation of static degree of use predictions used for studies of Section 3.3.3, the analysis

was also used in Section 4.1.2 to differentiate dead instructions and dynamically-dead instructions (both of which only generate useless instances).

For this work, the static dataflow analysis operated on compiled and linked object files. Because the degree of use dataflow problem pertains to values in architectural registers and would be performed after both register allocation and code generation, a suitable intermediate representation can be obtained from an object file. Also, modifying the compilers to perform the degree of use dataflow analysis during code generation would have been prohibitively time consuming and limited to the GNU compilers (gcc and g++) for which the compiler source code is available.

A.3.1 Precision considerations

The use of a binary analyzer does have consequences for the precision of the analysis. The possible targets of indirect branches and calls are not easily attainable from the object code. The analyzer instead uses the actual targets observed during an execution of the benchmark as the set of possible static targets of the indirect branch. Paths through indirect branches not actually exercised during the execution are not included in the analysis, leading to a more precise analysis than would be possible in a compiler.

Offsetting this advantage, the loss or obfuscation of other information within the binary leads to reduced precision. For example, statically-known loop counts are lost. Thus, the number of uses of results used within loop bodies cannot be determined, reducing the precision of the analysis. A similar situation exists with respect to dependent conditional statements wherein the analyzer does not know that certain paths through multiple conditionals are not actually possible. These effects reduce the precision of the binary analyzer relative to a compiler performing the same analysis.

A.3.2 Operation of binary analyzer

The binary analyzer operates as follows:

- **Generate control-flow graphs.** The disassembled object file is parsed and branch targets are recorded to determine basic block boundaries. Basic blocks are linked into per-procedure control-flow graphs.
- **Merge procedures connected by branches.** Certain procedures directly branch to other procedures when the implementation can be re-used. These procedures occur within the standard lan-

guage libraries when the implementation of the different procedures may be shared (e.g., the C library functions `memcpy()` and `memmove()`). Such procedures are merged into a single procedure to avoid analyzing the same instructions twice under different circumstances.

- **Generate an acyclic call graph.** The call graph contains one node per procedure and directed edges correspond to a call from one procedure to another. Strongly-connected components, which indicate the existence of cycles within the graph resulting from recursion, are detected using an implementation of Tarjan's algorithm [82] and collapsed into single nodes.
- **Compute ϕ-functions for each procedure.** A depth-first sort on the call graph orders the procedures such that all procedures appear before their callers (i.e., leaf procedures are processed first). ϕ-functions [75] are calculated at the basic block granularity. These functions convert facts true at the end of the procedure to facts true at the end of a basic block. Thus, the ϕ-function at the entry block of the procedure summarizes the dataflow effect of the entire procedure. The order in which procedures are processed ensures that the effects of all called procedures are known when generating the ϕ-function for the caller. Procedures belonging to strongly-connected components are iterated until the ϕ-functions converge.
- **Compute dataflow facts at each procedure exit.** In the reverse order that the ϕ-functions were generated (i.e., callers first), the facts at the end of the procedure are used to determine the facts true at each procedure call (with the ϕ-functions). For a given procedure, the facts true at the end of that procedure are just the meet (union) of the facts true after every call to that procedure within all calling procedures.
- **Annotate individual instructions.** The exit facts of each procedure together with the ϕ-function for a basic block within that procedure yield the facts true at the end of each basic block. This information is propagated backwards through the block to annotate each value-generating instruction with the possible degrees of use for that instruction, which is the final output of the analyzer.

A.4 Simulation

This section describes general attributes of the execution-driven simulation methodology used to obtain most of the results in this work. Details of the microarchitectural performance model will be presented in Section A.5.

A.4.1 Execution-driven simulation

Execution-driven simulation involves the emulation of each instruction executed on a *target* (simulated machine of interest) by the *host* (machine running the simulator). Modulo the accuracy of the emulation, the simulator is able to reproduce the changes in architectural state that would occur during an actual execution of a program on the target. In this work, the target is a machine implementing the Alpha instruction set architecture [5], including the BWX, CIX, and FIX, and MVI extensions that were introduced in later hardware implementations [4].

The simulators used in this work were built using components from the SimpleScalar v3.0 toolset [12]. SimpleScalar provided the underlying emulation code for the Alpha ISA and Digital UNIX system calls (see Section A.4.3), the program loader and memory space management (i.e., simulator to host address mapping) code, and some auxiliary niceties such as command-line parsing and basic statistics collection.

Significant changes were made to both the ISA and system call emulation code to fix errors exposed by the use of different compilation methods. Initially, many of the benchmarks did not run under the simulator or generated results that fell outside tolerances (see Section A.2.2) where a native run of the same binary yielded outputs within the tolerances. As a result, several new system calls and instructions had to be implemented and many existing implementations were corrected or enhanced. Most of these changes pertained to the precision and rounding modes of emulated floating-point operations.

A.4.2 Functional versus timing simulation

The minimalist execution-driven simulator, a *functional* simulator, tracks only the architected state. It models the target machine at the granularity of a single instruction, actually operating according to the von Neumann one-instruction-at-a-time model of program execution. Such a simulator is useful for the characterization of programs, such as those appearing in Chapter 2 and the beginning of Chapter 4, where neither timing information nor the effects of pipelining are of interest.

A *timing* or *performance* simulator models a specific implementation of the target at a much lower level. It adds a detailed, parameterized microarchitectural model, which simulates not only the results of executing an instruction, but how the instruction is executed in the modeled microar-

chitecture on a cycle granularity. Therefore, a timing simulator provides performance data (in terms of cycle count) as well as information about the behavior of the various components of the modeled microarchitecture during the execution of a benchmark. All performance evaluations in this work (and accompanying analyses) were performed using a timing simulator. In addition, the evaluation of the dynamic degree of use predictors of Section 3.4 was performed using timing simulation; while performance was not relevant to these experiments, the training of the dynamic predictor is affected by pipelining (see Section 3.4.6). The microarchitectural model assumed by the timing simulator is described in detail in Section A.5.

Because of the complexity of the timing simulator, it is about ten times slower than the functional simulator. Therefore, while benchmarks were executed to completion when functional simulation was used, only the first four billion instructions of each benchmark were executed in timing simulations. This particular sample of the benchmark is almost certainly not representative of the behavior of the entire benchmark in many cases. However, in no case was the true performance of a particular benchmark important—the goal was not to design a machine that delivered some level of performance on a workload based on the programs comprising the SPEC suite. Rather, the concern was that the benchmarks offer a wide variety of different behaviors under which the effects of particular microarchitectural adjustments can be evaluated. The data presented throughout this work indicate that this goal has been achieved.

A.4.3 System call emulation

The execution-driven simulation environment provided by the SimpleScalar toolkit handles a single user-level process only. In a real machine, system calls (to perform such tasks as I/O and system memory allocation) involve a transfer of control to the operating system and the execution of potentially privileged instructions (i.e., only available to the system software). In the single-process model of SimpleScalar, system calls are emulated by transferring parameters and (if necessary) data from simulated memory to host memory, executing the equivalent system call on the host, and transferring the results back to the simulated machine.

Emulating system calls in this manner influences the accuracy of the simulation in several ways. The entire operation of the system call, which may involve the execution of thousands of instructions, is condensed into a single atomic operation. From the perspective of the functional

simulator, these missing system instructions are the only real effect of system call emulation. For the types of studies performed with the functional simulator (benchmark characterization), the omission of system code is not important.

The effect of system call emulation on timing simulation is more profound. Scheduling and execution of other processes, hardware-generated interrupts, paging, and I/O accesses all affect microarchitectural state, which in turn impacts the performance of the benchmark. There is some evidence that the performance effect can be significant, even for the SPEC benchmarks, which spend relatively little time in system code [18]. Versus the results presented in this work, a real system would realize lower degree of use prediction accuracy (due to predictor interference by other processes) and lower overall performance (due to interference in other performance-enhancing structures, such as the caches and branch predictor). The relative performance impact of the optimizations presented would be correspondingly reduced as the benchmark performance was more influenced by memory stall time and branch mispredictions. The magnitudes of these various effects are difficult to estimate.

The actual handling of the system call by the simulator requires additional explanation. In the functional simulator, the system call is treated as a single instruction which executes to completion in isolation, like all other instructions. In the timing simulator, the detection of a system call (at decode) squashes all subsequent instructions and halts fetch. The machine completes all older in-flight instructions until the system call is the oldest (and only) instruction in the machine. At this point, the system call is emulated, which may involve the read and/or update of architected state. From the point of view of use tracking, a system call is treated as a single use followed by a new definition of every architected register.†

A.4.4 Static linking

Another consequence of the simulator's restriction to a single user-level program image is that it is incapable of calling upon the system's dynamic linker. As a result, programs are not able to make use of dynamically-linked shared libraries, such as the ubiquitous language standard librar-

---

† This accounting for system calls is not as arbitrary as it may seem. The Linux kernel [84], for example, adjusts the stack pointer prior to saving all integer registers to the stack, in effect using each register once (argument registers may see additional uses). The reverse process occurs on the completion of the system call resulting in a new definition for each register as its former value is loaded from the stack.

ies (e.g., `libc` and `libm` for C programs). Instead, all libraries must be statically-linked into a simulator executable.

The main consequence of static linking is that the compilers gain some additional visibility into the libraries and are able to perform inlining of library calls, code re-layout, and a few other post-link optimizations precluded by normal dynamic linking. More extensive optimizations are blocked because the libraries have already been compiled (when most optimization occurs) and their source code is not available during the benchmark compilation.

A.5 Timing Simulator Microarchitectural Model

The timing model used in this work was originally written by Craig Zilles of the University of Illinois while he was a graduate student at University of Wisconsin. His goal was the implementation of a highly-idealized microarchitectural model with which to investigate performance limitations inherent to programs executing on a superscalar machine (e.g., his work on performance-degrading instructions [94]). Over the past few years, most of the core microarchitectural model has been rewritten to reproduce limitations inherent to real hardware.

A block diagram of the microarchitecture modeled by the timing simulator appears in Figure A.1. The microarchitecture is representative of a modern out-of-order superscalar processor. The block diagram is divided into six modules for purposes of discussion. Each of the modules is shaded in the figure and labeled with the section in which that module is discussed.

The specifics of the microarchitecture are controlled by a large number of simulator parameters. Four different configurations of the timing simulator were used in the following evaluations: (1) dynamic degree of use prediction (Chapter 3), (2) useless instruction elimination (Chapter 4), (3) use-based register caching in an eight-wide machine (Chapter 5 except for Figure 5.11), (4) use-based register caching in a four-wide machine (Figure 5.11). The first of these was also used in the determination of the compiler flags yielding the highest performance (see Section A.2.2). The values of the simulator parameters are provided for each of these four configurations (tagged Ch. 3, Ch. 4, Ch. 5W, and Ch. 5N, respectively) in a table corresponding to each simulator module.

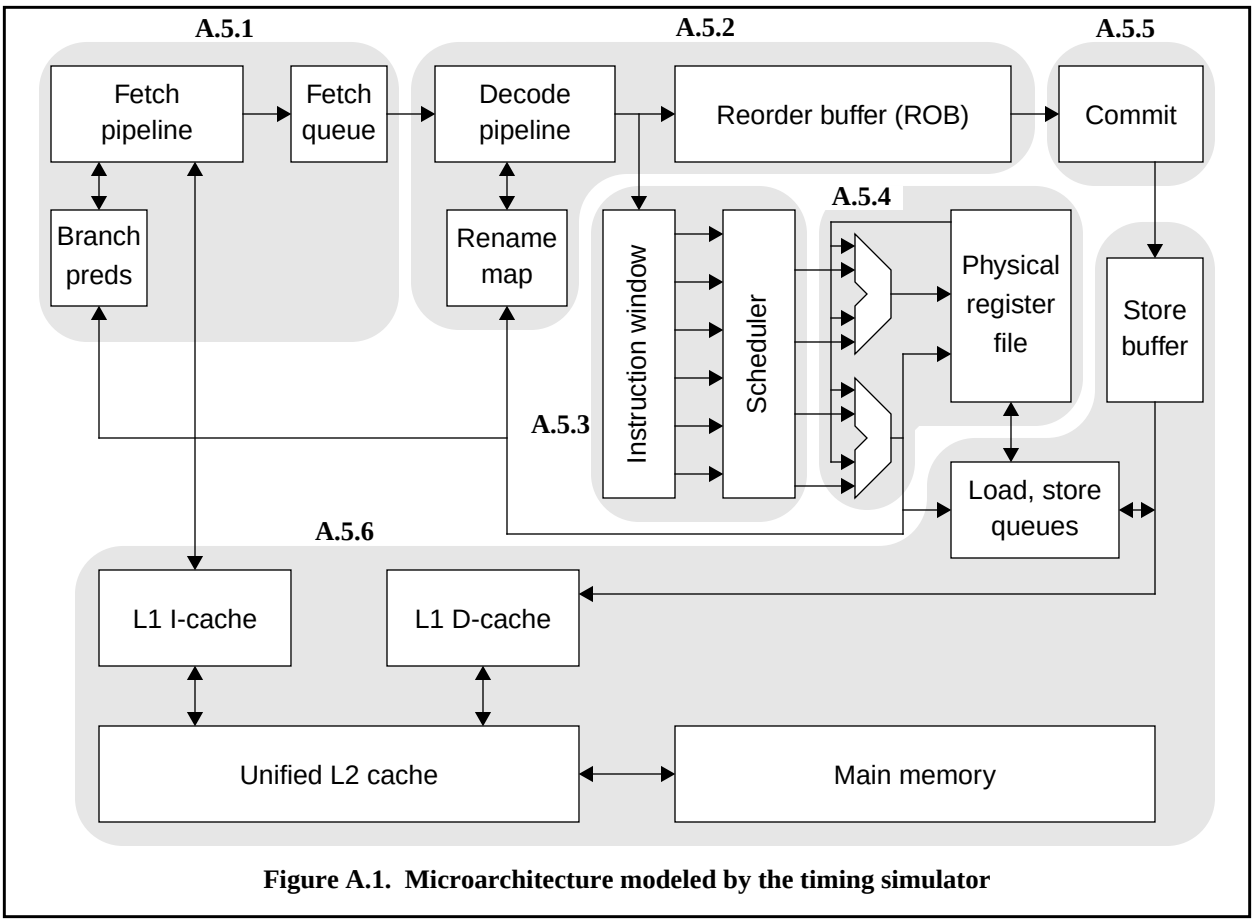


Figure A.1. Microarchitecture modeled by the timing simulator

A.5.1 Fetch pipeline (front end)

The fetch pipeline is responsible for instruction delivery. Each cycle that fetch is not stalled (e.g., due to a pipeline back-up, an instruction cache miss, or a bad fetch address), the front end injects a *fetch block* of instructions into the fetch pipeline, which models the latencies of generating the fetch address, accessing the instruction cache, and aligning the fetched instructions. A fetch queue decouples the fetch pipeline from the remainder of the machine, smoothing out variations in fetch block size.

The fetch width limits the maximum number of instructions in a fetch block. NOPs are eliminated from the instruction stream here and do not count against the fetch block size (although they increase the probability that a fetch block will reach a cache line boundary). As in the Alpha 21264, conditional move instructions—the only three-input instructions—are cracked into two data-dependent, two-input instructions [4]. These are treated independently from fetch through

Table A.3: Fetch Pipeline Parameters

Parameter	Ch. 3	Ch. 4	Ch. 5W	Ch. 5N
Non-NOP instructions fetched/cycle (fetch block size)	8	4	8	4
Instruction cache read ports	2	1	2	1
Stages in fetch pipeline (includes I-cache latency)	6	3	5	5
Maximum taken control instructions per fetch block	2	1	2	1
Entries in fetch queue	56	16	48	24
YAGS branch predictor: history bits ($\log_2$ PHT entries)	14			
YAGS: T/NT exception table entries (each)	4096			
YAGS: exception table tag bits (branch_PC[7:2])	6			
Cascaded indirect branch predictor: simple table entries	256			
Indirect: exception table entries	1024			
Indirect: history path length (see footnote on page 192).	3			
Return address stack (RAS) entries	64			

retirement (e.g., each one consumes an instruction slot in the fetch block). Only those statistics based on retired instruction count (e.g., IPC), count the pair as a single instruction.

The fetch block may contain fewer instructions than the fetch width. The total number of taken control instructions (of any kind) is limited; fetch blocks may include any number of non-taken conditional branches. Instruction cache access ports are modeled by limiting the number of cache lines spanned by the fetch block. If the flow of control leaves and returns to a particular cache line, the block is treated as a new line.

Conditional branches are predicted serially using a YAGS [27] branch predictor. The global branch history register is modified speculatively with each branch prediction; it is recovered and corrected (if necessary) on exceptions. The target addresses of taken conditional branches are assumed to be available (i.e., a perfect BTB). The target addresses of indirect jumps and calls (except for returns) are predicted using a cascaded indirect branch predictor [26] with leaky filtering.† The target of a procedure return is obtained by popping a return address stack [88] on which each direct and indirect call pushes the address of its subsequent instruction. The top-of-stack pointer is recovered on exceptions [76].

A.5.2 Decode pipeline

Instructions from the fetch queue are renamed and allocated resources in the decode pipeline. The depth of this pipeline models the latency of these operations, although, as in the front end, they occur logically in the first stage. Entry into the decode pipeline is gated by the availability of the allocated resources, which include instruction window and reorder buffer entries, physical registers, and load or store queue (LSQ) entries. Except for LSQ entries, the lack of enough of each resource to handle one cycle of full-width decoding will block *any* instructions from entering the decode pipeline; a full load or store queue stalls the decode pipe at the first instruction needing it.

Instructions are renamed by mapping their source architectural registers to physical registers via a RAM lookup as in the MIPS R10K [89]. Those instructions that write a register are allocated a free physical register from a stack. The state of the rename map is speculative and is recovered on a branch misprediction or other exception condition. The microarchitectural model assumes that the map state prior to the rename of any unretired instruction can be recovered (i.e., no limit on the number of in-flight speculative branches). These events are always associated with a re-fetch of instructions from the recovery point, and the map is assumed to be corrected by the time the first re-fetched instruction reaches the renamer (i.e., no extra recovery latency).

In the simulator model, the dynamic degree of use predictor logically exists completely within the first decode stage of the pipeline (together with the rest of the decode pipeline operations), unlike in the depiction in Figure 3.6. Thus, the predictor is only accessed for instructions reaching the renamer instead of all fetched instructions, and the resulting predictions are available immediately. This does not affect the accuracy results presented in Chapter 3, since prediction coverage and accuracy are calculated only on instructions that retire. It does, however, mask a mismatch between the fetch bandwidth (which corresponds to the predictor access bandwidth) and the rename bandwidth (which corresponds to the prediction consumption bandwidth) due to the presence of the fetch queue. An actual implementation could handle this problem

---

† The simple table is indexed with PC[9:2] ⊕ PC[17:10] ⊕ PC[25:18], where PC is the indirect branch PC. The exception table is indexed with PC[11:2] ⊕ PC[21:12] ⊕ PC[31:22] ⊕ compressed_history[9:0]. The compressed history is 0.0.targhist[7:0] ⊕ 0.targhist[15:8].0 ⊕ targhist[23:16].0.0. Each byte of the (global) target history corresponds to a prior indirect branch target. Thus, three prior indirect branch targets are used in generating the exception table index. The target history is updated by shifting left one byte and placing target_PC[9:2] ⊕ target_PC[17:10] in the least significant byte. For details on the roles of the different tables, refer to the paper describing the cascaded indirect predictor [26].

Table A.4: Decode Pipeline Parameters

Parameter	Ch. 3	Ch. 4	Ch. 5W	Ch. 5N
Instruction decode bandwidth (decode IPC)	8	4	8	4
Stages in decode pipeline	5	3	5	5
Reorder buffer (ROB) entries	512	256	320	256

by: (1) adding a separate FIFO queue for the degree of use predictions, (2) adding a write port into the fetch queue for the predictions, or (3) ensuring that the predictions were generated prior to writing the corresponding instruction into the fetch queue.

A.5.3 Instruction window and scheduler

Instructions pass from the decode pipeline into the instruction window. Each cycle the instruction scheduler selects instructions from the window for issue to the execution pipelines. An instruction is eligible for issue when its input operands are ready, there is an unused issue port available with the appropriate execution resource, and the issue port's writeback bus is free in the cycle that the instruction will complete. Memory instructions have additional requirements: a load or store requires a free L1 MSHR (in case of a data cache miss), and a load may be delayed until certain older stores that could bypass to that load have computed their addresses (see Section A.5.6). The instruction window selects the oldest eligible instruction until no more instructions are eligible for issue or the peak issue bandwidth (equal to the number of issue ports) is reached.

The partitioning of the execution resources among issue ports is an important determinant of the issue behavior. During any given cycle, only one instruction may be issued on a particular port, which blocks not only the corresponding resource, but all other execution resources bound to the same port. Each execution resource is assumed to be fully-pipelined (i.e., it can accept a new operation each cycle).

An instruction's input operand is ready if that operand will be available from either the bypass network or the register file by the time the instruction would need it for execution. If the register file is not fully-bypassed, a register result may become ready on the bypass network upon instruction completion, then become unready again prior to the completion of the register file write (refer Figure 5.4(a) and the accompanying discussion).

Table A.5: Instruction Window and Scheduler Parameters

Parameter	Ch. 3	Ch. 4	Ch. 5W	Ch. 5N
Instruction window entries	128	64	200	128
Issue port 1[a]	N/A[b]	I[c]	I	I
Issue port 2		IM	LS	IM
Issue port 3		FYDR	LS	ILS
Issue port 4		L	IF	LS
Issue port 5		SB	IF	B
Issue port 6		–	IMBFYR	F
Issue port 7		–	IMBFYR	FYDR
Issue port 8		–	ILS	–

a. Simple (I)nteger, integer (M)ultiply, integer (B)ranch, simple (F)loating-point, FP multipl(Y), FP (D)ivide and square root, FP b(R)anch, (L)oad, (S)tore.
b. An earlier issue model was used that did not model the binding of execution resources to specific issue ports. For these studies, *any* combination of eight instructions could issue each cycle to the following resources: 6×I, 2×M, 2×B, 3×F/R, 3×Y/D, 3×L, 3×S.
c. This issue configuration represents the medium resource model from Chapter 4. See Table 4.4 for details on other issue configurations used in that chapter.

Operand readiness is speculative: loads are assumed to hit in the L1 data cache and, where a register cache is used, input operands are assumed to be present in the register cache. A data cache or register cache miss on an issued instruction implies that the speculative readiness information is incorrect and results in the reissue (replay) of all instructions issued after the instruction causing the replay. Instructions continue to occupy the instruction window after issue until they are known to be replay-safe. Instructions become replay-safe a fixed latency after issue when the register or data cache hit-miss status has been determined. This speculative scheduling model is similar to that implemented by the Alpha 21264 [4, 48].

A.5.4 Register file and execution

Issued instructions read their input operands from the register file beginning in the cycle after they are issued. The register file read may span multiple cycles (see Figure 5.4(a)) and occurs regardless of whether an operand is eventually obtained from the bypass network. A register cache is simply treated as a single cycle register file for purposes of the execution pipeline. When a miss is detected, subsequently issued instructions are replayed, and the appropriate penalties are added to the execution latency of the instruction incurring the miss (see Section 5.2).

Table A.6: Register File and Execution Parameters

Parameter	Ch. 3	Ch. 4	Ch. 5W	Ch. 5N
Physical register file entries (shared integer & FP)	512	256	320	256
Register file read latency (cycles)	1	1	3	3
Register file write latency (cycles)	1	1	3	3
Bypass network stages	2	2	2	2
Simple integer operation latency (cycles)	1			
Integer multiply latency (cycles)	4			
Branch execution latency (cycles)	2			
Simple floating-point operation latency (cycles)	2			
Floating-point multiply latency (cycles)	4			
Floating-point divide latency (cycles)	16			
Floating-point square root latency (cycles)	33			
Load-to-use (L1 hit) and L1 miss detect latency	4	3	3	3
Store latency (to detect collisions and bypass to loads)	3	2	2	2

After completion of the register read, a number of cycles equal to the execution latency is counted. In the final cycle, the ISA implementation code (shared with the functional simulator) performs the required operation on the actual input data (which is speculative and may be incorrect). Memory operations check the LSQ and data cache as described in Section A.5.6.

A.5.5 Commit

The commit logic operates as an in-order pipeline decoupled from the rest of the machine. The oldest in-flight instruction (at the head of the reorder buffer) is checked for retirement eligibility. An instruction becomes eligible for retirement after it has executed and the writeback of the instruction's result into the register file (where necessary) has completed (i.e., the write latency of the register file affects the earliest commit time). Eligible instructions are retired in program order up to a maximum retirement bandwidth.

Store retirement is also gated by the availability of a cache write port and a free store buffer entry. Thus, the maximum number of stores retired per cycle equals the cache write bandwidth. The use of a store buffer implies that stores need not complete the cache write operation to retire

Table A.7: Commit Parameters

Parameter	Ch. 3	Ch. 4	Ch. 5W	Ch. 5N
Instruction commit bandwidth (commit IPC)	8	4	8	4
Maximum stores retired per cycle	3	1-2[a]	3	2

a. Equal to the store execution bandwidth for the issue model used. See Table 4.4 for details.

and allows retirement to proceed past store misses. Stores are drained from the store buffer in program order as the cache write operations complete.

A.5.6 Memory system

Independent load and store queues are responsible for maintaining memory dependences defined by program order and for communicating memory values among speculative, in-flight instructions. Entries are allocated to loads and stores in program order in the decode pipeline (see Section A.5.2). Load entries are freed when the corresponding instruction commits while store entries are freed after the store data is written to the cache. When a load or store executes, its physical address is known (a perfect TLB is assumed) and used to probe the opposite queue by address.

A load searches the store queue for the youngest older store to the same address. If such a store is found, the load result is bypassed from the store data; otherwise, the data is obtained from the cache. Address matching is performed at a 64-bit granularity. If a matching store supplies only part of the load data (a partial overlap), older stores may also be included. In the extreme case, a 64-bit load may bypass from eight 8-bit stores. The ability to perform *partial bypassing* was not present in earlier versions of the simulator, which could only handle loads bypassing from a single store *or* the cache. In this case, partial overlaps resulted in a load replays (see below). Store address and store data operations are not distinguished (i.e., store address operations do not issue independently of store data operations); therefore, a store's data is always available once its address is known and a load will never have to wait for the data of a known-matching store.

Each store probes the load queue for a younger load that should have bypassed from the store but obtained its result from the cache or from an older store than the store under consideration. This results in a load-dependence replay wherein the load and all younger instructions are squashed and fetch resumes with the load.

Table A.8: Memory System Parameters

Parameter	Ch. 3	Ch. 4	Ch. 5W	Ch. 5N
Load queue entries	128	64	128	128
Store queue entries	128	64	128	128
Partial bypassing	no	yes	yes	yes
Load dependence predictor entries (direct-mapped)	64			
L1 cache[a] capacity (2-way set assoc.; KB)	32	64	32	32
L1 block size (bytes)	64			
L1 MSHRs	64			
L1 stream buffers	8			
L1 stream buffer size (blocks)	4			
L1-L2 bus bandwidth (bytes/cycle)	16			
L2 cache capacity (unified; 4-way set assoc.; KB)	1024	2048	2048	2048
L2 block size (bytes)	128			
L2 MSHRs	64			
L2 stream buffers	16			
L2 stream buffer size (blocks)	4			
L2-memory bus bandwidth (bytes/cycle)	8			
L2 latency (cycles)	12	8	12	12
Memory latency (cycles)	180	100	160	160
Store buffer entries	16			

a. L1 instruction and data caches are identical and independent.

A load dependence predictor minimizes the occurrence of expensive load replays by delaying the issue of loads with a history of causing these events. The predictor relies upon a table of collision distances [90], which is written in the event of a load-dependence replay. The table is indexed and tagged by the PC of the load instruction and contains the number of consecutive older stores (beginning with the youngest older store) that did *not* collide with the load. This distance is used to determine when subsequent instances of the load should issue. Since stores within this distance are not expected to bypass to the load, their addresses do not need to be known prior to issuing the load. Any store further away (older) than the collision distance must have issued and computed its address before the load can issue.

A store buffer queues committed stores until their cache write is complete. The store buffer is emptied into the L1 data cache in program order. A store miss stalls the processing of stores in the buffer until the appropriate line is fetched into the L1 data cache. Data is bypassed from the store buffer on a cache read access such that loads retrieve the most-recently-written data.

The memory hierarchy consists of separate, identical L1 instruction and data caches and a unified L2 cache over an infinite, fixed-latency main memory. The cache hierarchy is writeback and inclusive. Each cache has MSHRs that track outstanding misses and prefetches, allowing accesses to occur in parallel with resolving misses (i.e., the caches are non-blocking [54]). Each cache also employs an opportunistic stride-based prefetcher [34] that generates hardware prefetches for multiple, potentially interleaved streams into associated stream buffers [47]. Buses connecting different levels of the cache hierarchy with each other and memory are modeled and bandwidth-limited. Writebacks and fills schedule the bus and block its use for an appropriate number of cycles to transfer the necessary data. Data transfer occurs critical-word-first (e.g., for a fill, the requested word within the line is transferred across the bus first, regardless of its offset within the line).