Compiling for the Multiscalar Architecture

by

T. N. Vijaykumar

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

1998

Abstract

High-performance, general-purpose microprocessors serve as compute engines for computers ranging from personal computers to supercomputers. Sequential programs constitute a major portion of real-world software that run on the computers. State-of-the-art microprocessors exploit instruction level parallelism (ILP) to achieve high performance on such applications by searching for independent instructions in a dynamic window of instructions and executing them on a wide-issue pipeline. Increasing the window size and the issue width to extract more ILP may hinder achieving high clock speeds, limiting overall performance.

The Multiscalar architecture employs multiple small windows and many narrow-issue processing units to exploit ILP at high clock speeds. Sequential programs are partitioned into code fragments called tasks, which are speculatively executed in parallel. Inter-task register dependences are honored via communication and synchronization and inter-task control flow and memory dependences are handled by speculation and verification in hardware.

Since this thesis is the first attempt at investigating the problem of compiling for the Multiscalar architecture, I identify the fundamental interactions between applications and the Multiscalar architecture from the standpoint of performance and explore a few compiler optimization opportunities instead of proposing the best technique for a specific problem.

Control flow speculation, register communication, memory dependence speculation, load imbalance, and task overheads are key performance issues. To extract high degrees of ILP, compiler heuristics partition programs into large tasks, which comprise multiple basic blocks. To maintain high prediction accuracy and avoid delays due to inter-task register communication, the heuristics control the number of successors of tasks while including register dependences within tasks. Inter-task register communication is generated and scheduled to overlap computation and inter-task register communication.

For the SPEC95 benchmarks, the heuristics increase task sizes significantly while improving control flow speculation accuracy with respect to basic blocks, enabling large window spans from which to extract parallelism. Including register dependences within tasks improves performance considerably. Sophisticated register communication generation and scheduling are effective in boosting performance. Dead register analysis reduces register communication traffic considerably. All the optimizations grow in importance for larger number of PUs.

கற்றது கை மண்ணளவு கல்லாதது உலகளவு

ஒளவையார்

Acknowledgments

I thank my advisor, Prof. Guri Sohi, for his patience and guidance. He gave me the opportunity to work on the Multiscalar project and fostered an environment in which I could pursue my ideas unfettered. He taught me how to write and how to do research in computer architecture. Throughout my graduate career, I could candidly express my concerns and opinions, both technical and nontechnical, and he was always receptive, even when we disagreed.

I thank my parents, Sri. T. N. Gopalachari and Smt. Choodamani Gopalan, for instilling the value of education in me. They taught me that hard work and dedication are necessary ingredients for success in any endeavor. They put me through college and sent me to the US for graduate education. I thank them for their support, sacrifice and trust. I thank my sister, Smt. Mythreyi Srinivasan, for always being there for me.

I thank Prof. Jim Smith for many insightful observations on the Multiscalar architecture and for incisive feedback on my research. I thank Prof. Mark Hill for stressing the importance of good presentation and for invaluable advice on how to succeed in academe at all stages of my career, including the future. I thank Prof. David Wood for his technical insight into various aspects of my research topic and Prof. Jim Goodman for his enlightening perspective on computer architecture and graduate school.

I thank Profs. Guri Sohi, Jim Smith, Mark Hill, David Wood and Jim Goodman for teaching me computer architecture.

I thank my partner-in-crime, Scott Breach, for his wonderful simulator, without which I could not have done this research. Not once did Scott decline my request for changes in his simulator to help debug my compiler. Without Scott's thoroughness and uncompromising standard of quality, the simulator would not be robust enough to withstand the assault of code generated by my compiler. I thank my office-mate, Andreas Moshovos, for setting up the benchmarks and the compile-scripts used in my research. I thank Sridhar Gopal for his contributions to the simulator. I thank Scott, Andreas and Sridhar for many productive discussions on the Multiscalar architecture.

I thank Scott Breach, Doug Burger, Babak Falsafi, and Alain Kagi for their irreplaceable friendship. Their endearing camaraderie made my professional and personal life enjoyable, thousands of miles away from home, in a foreign country. I learned many things about computer architecture and life in the US from them. I am indebted to them for their unflinching support throughout my graduate career in Wisconsin.

I thank Po-Tai Cheng, Steve Kurlander, Subbarao Palacharla, Jignesh Patel, and Francis Valiyaveetil for sharing a house with me at different points of my stay in Madison. Their bonhomie enriched my life and made mundane chores pleasurable. I thank Shubu Mukherjee for many discussions during lunches and Satish Chandra and Stefanos Kaxiras for many interesting conversations. I thank Manoj Franklin for answering my questions on the Multiscalar architecture when I joined the project. I thank Dionisios Pnevmatikatos and Todd Austin for many discussions on computer architecture. I thank Harish Patil, Ambuj Shatdal, and Shivakumar Venkataraman for their friendship in the early days of my graduate career.

Finally, I thank Richard Stallman and the people who contributed to the Gnu C compiler, gcc. My research benefited greatly from their stupendous effort.

Table of Contents

Abstract i

Acknowledgments iv

List of Figures x

List of Tables xiii

Chapter 1. Multiscalar: A Novel Microarchitecture 1

1.1 Performance limitations of superscalar processors 4

1.2 Multiscalar approach 5

1.3 Thesis contributions 8

1.3.1 Task selection 9

1.3.2 Inter-task dependence information 9

1.3.3 Register communication 10

1.4 Thesis roadmap 10

Chapter 2. Role of the Hardware and the Compiler 12

2.1 Basic concepts of the Multiscalar architecture 12

2.2 Multiscalar tasks 15

2.3 execution model for the Multiscalar architecture 15

2.3.1 Control flow 16

2.3.2 Data values 17

2.3.3 Sequential semantics 22

2.4 Application binary interface 23

2.4.1 Inter-task control flow information 24

2.4.2 Inter-task data dependence information 24

2.4.3 An example of a Multiscalar program 26

2.4.4 Other alternatives to binary interface 29

2.5 Hardware implementation of the Multiscalar architecture 29

2.5.1 Control flow: Sequencer 31

2.5.2 Register data: Unidirectional communication ring .31

2.5.3 Memory data: Address resolution buffer 32

2.5.4 Sequential semantics 32

2.6 Performance Aspects 32

2.6.1 Scenario 1: Task is retired 33

2.6.2 Scenario 2: Task is squashed 33

2.6.3 Performance improvements through the compiler .35

2.7 The problem of compiling for the Multiscalar architecture 36

2.7.1 Correctness issues 38

2.7.2 Performance issues 39

2.7.3 Organization of the compiler 39

Chapter 3. Task Selection 41

3.1 Definition 43

3.2 Performance issues and task characteristics 46

3.2.1 Control flow misspeculation 46

3.2.2 Inter-task data dependence 48

3.2.3 Load imbalance 51

3.2.4 Task overheads 52

3.2.5 Summary: Task characteristics that affect performance 53

3.3 Task selection criteria 54

3.3.1 Task size criteria 54

3.3.2 Control flow criteria 55

3.3.3 Data dependence criteria 58

3.3.4 Interaction among the criteria 62

3.4 Task selection heuristics 63

3.4.1 Basic task selection process 64

3.4.2 Control flow heuristic 64

3.4.3 Data dependence heuristic 65

3.4.4 Special heuristics 70

3.4.5 Short function invocations 70

3.5 Implementation of task selection 73

3.5.1 Implementing control flow heuristics 73

3.5.2 Implementing data dependence heuristics 74

3.5.3 Code replication 77

3.5.4 Code layout 78

3.6 Related work 78

Chapter 4. Register Communication 81

4.1 Register communication model 82

4.2 Strategies for orchestrating register communication 85

4.2.1 End_send and eager_send 88

4.2.2 Last_send 89

4.2.3 Spec_send 91

4.3 Implications of dead registers and register assignment 93

4.3.1 Dead registers 93

4.3.2 Register assignment 95

4.4 Register communication scheduling 96

4.4.1 Scheduling strategy 98

4.4.2 Identifying the instructions for scheduling 100

4.4.3 Code motion 100

4.4.3.1 Issues related to code motion 103

4.4.4 Cost model 104

4.4.5 Controlling instruction count increase 108

4.4.6 Scheduling loop induction variables 110

4.4.7 Interaction with other optimizations 110

4.5 Implementation 112

4.6 Related work 114

Chapter 5. Experimental Evaluation 118

5.1 Overview of experiments 119

5.2 Experimental methodology 121

5.2.1 Framework 122

5.2.2 Simulation parameters 122

5.2.2.1 Processing units 122

5.2.2.2 Control flow prediction 123

5.2.2.3 Register communication 123

5.2.2.4 Memory subsystem 123

5.2.2.5 Compiler optimizations 125

5.2.3 Benchmarks 127

5.3 Experiments 127

5.3.1 Effectiveness of compiler heuristics 130

5.3.2 Task characteristics and metrics 135

5.3.2.1 Task size metrics 136

5.3.2.2 Inter-task control flow metrics 143

5.3.2.3 Inter-task data dependence metrics 148

5.3.3 Window span 156

5.3.4 Experiments related to register communication 161

5.3.4.1 Effectiveness of register communication strategies 162

5.3.5 Impact of register communication scheduling 165

5.3.5.1 Overhead of release instructions 170

5.3.5.2 Effectiveness of dead register analysis 171

Chapter 6. Conclusions 173

6.1 Thesis summary 174

6.2 Thesis conclusions 176

6.3 Future work 180

List of Figures

Figure 1-1: Extraction of ILP in the superscalar architecture 3

Figure 1-2: Extraction of ILP in the Multiscalar architecture 6

Figure 1-3: Mapping a sequential program on the Multiscalar architecture 7

Figure 2-1: Abstraction of the Multiscalar architecture. 13

Figure 2-2: Control flow speculation in the Multiscalar architecture 17

Figure 2-3: Register dependences and memory dependences 18

Figure 2-4: Register Communication in the Multiscalar Architecture 20

Figure 2-5: Memory disambiguation in the Multiscalar architecture 21

Figure 2-6: An example of a Multiscalar program 27

Figure 2-7: Hardware for the Multiscalar architecture 30

Figure 2-8: Time line of the execution of a task 34

Figure 2-9: Role of the compiler for the Multiscalar architecture. 37

Figure 2-10: Organization of the compiler for the Multiscalar architecture 40

Figure 3-1: Examples of Multiscalar tasks 45

Figure 3-2: Impact of control flow. 47

Figure 3-3: Impact of data dependences 49

Figure 3-4: Number of successors 56

Figure 3-5: Task selection and data dependence edges 60

Figure 3-6: Task selection and inter-task data communication scheduling. 61

Figure 3-7: Including a data dependence within a task 67

Figure 3-8: Data dependence heuristics. 68

Figure 3-9: Effect of loop recurrences. 71

Figure 3-10: Control flow heuristics 74

Figure 3-11: Data dependence heuristics. 75

Figure 3-12: Code replication during task selection 77

Figure 3-13: Code layout after task selection 79

Figure 4-1: Register communication model 84

Figure 4-2: An example illustrating the four strategies 87

Figure 4-3: Dataflow equations for last_send 90
Figure 4-4: Dataflow equations for spec_send. 92
Figure 4-5: Register communication annotation.. 93
Figure 4-6: An example of register assignment delaying register forward. 96
Figure 4-7: Register communication dynamics 97
Figure 4-8: Run time dynamics of the example. 99
Figure 4-9: Upward transformations.. 101
Figure 4-10: Downward transformations. 102
Figure 4-11: Cost model 105
Figure 4-12: An example of register communication scheduling.. 107
Figure 4-13: Conditions to prevent unnecessary dynamic instruction increase. 109
Figure 4-14: Induction variables in loops. 111
Figure 4-15: Pseudo-code for register communication generation 113
Figure 4-16: Pseudo-code for scheduling register communication 115
Figure 5-1: Impact of compiler heuristics on CINT benchmarks with out-of-order PUs132
Figure 5-2: Impact of compiler heuristics on CFP benchmarks with out-of-order PUs. 133
Figure 5-3: Impact of compiler heuristics on CINT benchmarks with in-order PUs . . 134
Figure 5-4: Impact of compiler heuristics on CFP benchmarks with in-order PUs. . . . 136
Figure 5-5: Observed load imbalance stall for SPEC CINT benchmarks. 141
Figure 5-6: Observed load imbalance stall for SPEC CFP benchmarks 142
Figure 5-7: Observed task overhead stall for SPEC CINT benchmarks.. 144
Figure 5-8: Observed task overhead stall for SPEC CFP benchmarks 145
Figure 5-9: Number of targets for SPEC95 CINT benchmarks. 148
Figure 5-10: Number of targets for SPEC95 CFP benchmarks 149
Figure 5-11: Observed register dependence stall for SPEC CINT benchmarks 150
Figure 5-12: Observed register dependence stall for SPEC CFP benchmarks 151
Figure 5-13: Observed memory dependence stall for SPEC CINT benchmarks. 153
Figure 5-14: Observed memory dependence stall for CFP benchmarks 154
Figure 5-15: Inter-task register dependences for CINT benchmarks. 155

Figure 5-16: Inter-task register dependences for CFP benchmarks 156
Figure 5-17: Inter-task memory dependences for CINT benchmarks 157
Figure 5-18: Inter-task memory dependences for CFP benchmarks 158
Figure 5-19: Window span of CINT benchmarks for data dependence tasks 159
Figure 5-20: Window span of CFP benchmarks for data dependence tasks 160
Figure 5-21: Window span of CINT benchmarks for control flow tasks 161
Figure 5-22: Window span of CFP benchmarks for control flow tasks 162
Figure 5-23: Window span of CINT benchmarks for basic block tasks 163
Figure 5-24: Window span of CFP benchmarks for basic block tasks 164
Figure 5-25: Register communication strategies for CINT benchmarks 165
Figure 5-26: Register communication strategies for CFP benchmarks 166
Figure 5-27: Register communication scheduling for CINT benchmarks 167
Figure 5-28: Register communication scheduling for CFP benchmarks 168
Figure 5-29: Register communication scheduling for CINT benchmarks 169
Figure 5-30: Register communication scheduling for CFP benchmarks 170

List of Tables

Table 5-1: Hardware parameters used in experimental evaluation. 126

Table 5-2: Inputs for CINT benchmarks. 128

Table 5-3: Inputs for CFP benchmarks. 129

Table 5-4: Dynamic task size for CINT benchmarks. 137

Table 5-5: Dynamic task size for CFP benchmarks. 138

Table 5-6: Control Flow Misspeculation Rate for SPEC95 CINT Benchmarks 146

Table 5-7: Control flow misspeculation rate for SPEC95 CFP benchmarks. 147

Table 5-8: Overhead of dynamic release instructions for SPEC95 benchmarks. 171

Table 5-9: Impact of dead register optimization on SPEC95 benchmarks. 172

Chapter 1

Multiscalar: A Novel Microarchitecture

High-performance, general-purpose microprocessors serve as cost-effective compute engines for a wide spectrum of computers, ranging from the ubiquitous personal computer to state-of-the-art, esoteric parallel computer. Software that run on microprocessors represent a wide variety of application areas such as business and engineering, science and education, home and entertainment, most modern industries and government. These application areas have continually placed an ever-increasing demand on performance of microprocessors. Improving performance of microprocessors not only enhances the utility of existing application domains by accelerating the computations involved but also enables the emergence of new application domains by enabling computations, which were considered infeasible in the past.

Sequential programs constitute a major portion of software that run on microprocessors powering the wide spectrum of computers. Ease of programming and transparent portability with high performance have been compelling reasons for the pervasiveness and the magnitude of sequential programs in application software domains. The rest of the software are multithreaded or parallel applications that run on microprocessors used in large-scale, parallel computers. Even for parallel applications, single sequential thread performance is important to avoid the performance limits stipulated by Amdahl's law [7]. So, irrespective of the application domain and the scale of the computer, it is critical to achieve high performance on sequential programs that run on microprocessors.

Execution of a sequential program on a processor is realized as a single stream of instructions that manipulates data according to the computation specified by the program. This stream of instructions is called the **dynamic instruction stream**. In order to improve performance of the program, processor architects have exploited independence among the instructions in the dynamic instruction stream [8] [31] [91] [100] [101] [102] [94]. **Instruction level parallelism (ILP)** [35] advocates the execution of such independent instructions in parallel. ILP is an implicit form of parallelism present in sequential programs identified by a combination of the compiler and the hardware, whereas explicit parallelism of parallel programs is identified and specified by the programmer.

To exploit ILP, modern microarchitectures called **superscalar architectures** [11] [21] [29] [44] [46] [47] [53] [58] [77] [97] [109] [113] identify independent instructions and execute multiple independent instructions simultaneously. Modern microprocessors extract ILP by establishing a **window** of instructions in the dynamic instruction stream. The first instruction in the window is called the **point of search** for the window. The window is said to be established at the point of search in the dynamic instruction stream. Within the window, superscalar architectures search for independent instructions by determining dependence relationships among the instructions. Multiple independent instructions are executed simultaneously by a set of hardware resources. The maximum number of instructions that may be executed in parallel is called the **width** of the processor. The maximum number of instructions that may be contained in the window is called the **size of the window**. Figure 1-1 illustrates extraction of ILP by superscalar architectures. The degree of parallelism extracted depends on the size of the window and the width of the processor. Larger windows enable more dynamic instructions to be examined, which leads

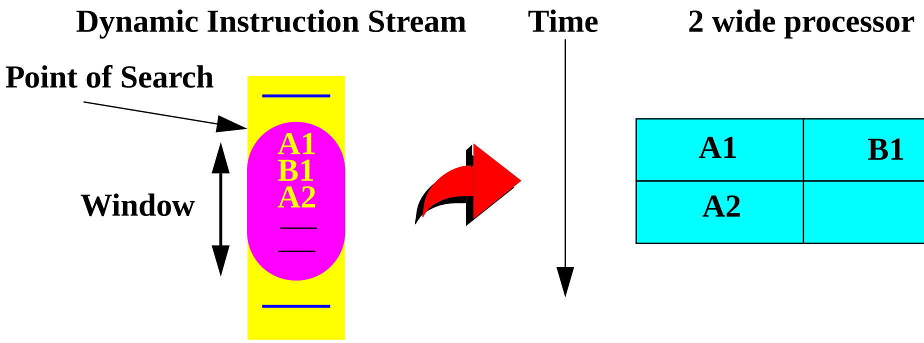


Figure 1-1: Extraction of ILP in the superscalar architecture. The window established at the point of search contains instructions A1, B1, andA2. A2 is dependent on A1. A1 and B1 execute simultaneously on a 2-wide processor and at a later clock cycle, A2 executes.

to the identification of more independent instructions and wider processors can execute more independent instructions simultaneously. But, larger windows and wider processors may be harder to engineer at high clock speeds, limiting performance.

The **Multiscalar** architecture [36] [38] [95] is a novel microarchitecture to achieve high-performance on general-purpose, sequential programs. To engineer a large window and a wide processor at high clock speed, the Multiscalar architecture splits one wide processor into many narrow processing units and one large window into many small windows monitored by each processing unit. Sequential programs are partitioned into code fragments, which are executed on the processing units in parallel. Dependences among the code fragments are honored by a combination of communication and synchronization for dependences that are known at compile-time or speculation and verification for dependences that are ambiguous at compile-time.

The goals of this thesis are: (1) to identify the fundamental interactions between applications and the architecture from the standpoint of performance, (2) to understand the implications of the interactions for the compiler, and (3) to devise, implement, analyze and experiment with compiler techniques to improve performance. Since this thesis is the first attempt at investigating this problem, I identify the key issues involved in the problem of compiling for the Multiscalar architecture and explore a few compiler optimization opportunities instead of proposing the best technique to solve a specific optimization problem. To make the investigation concrete, I constructed a compiler and evaluated the impact of compiler techniques on performance of the SPEC95 benchmark suite [98].

1.1 Performance limitations of superscalar processors

Overall performance is the inverse of the product of the parallelism extracted and the cycle time of the processor. State-of-the-art microprocessors exploit high degree of parallelism by increasing the size of the window and the width of the processor. But larger window and wider processor may exacerbate circuit delays; implying that more parallelism may be extracted at the cost of clock speed. Although increasing the window size and the processor width may increase the degree of parallelism extracted, overall performance may not improve or may even deteriorate. Thus, this straightforward approach may be forced to trade-off clock speed for parallelism.

Processor clock speed is determined by the timing constraints of the circuits that are in the critical path of the processor pipeline implementation. Timings of many critical paths such as rename logic, wake-up logic, and bypass logic [80] in processor pipeline hardware are at least linear, if not quadratic, functions of window size and processor width for the circuit implementations used by state-of-the-art microprocessors. These circuits cannot easily be pipelined to be clocked faster without incurring pipeline stall cycles.

The continual shrinking of feature size of on-chip devices in modern microprocessors enable faster and smaller transistors, leading to phenomenal improvements in processor clock speeds. Unfortunately, wires connecting the transistors do not scale in speed as well as the transistors themselves because quadratic RC delays dominate total signal delay [16]. The centralized design of the superscalar architectures imposes lengthening of the wires in critical circuits like bypass paths and wakeup logic, at least by a factor linearly proportional to window size and processor width [80]. Long wires connecting many devices for bypassing data values and for performing result update in the superscalar pipeline datapath may limit clock speed. If time taken to propagate a signal through a wire is longer than that for a transistor to switch, a signal may take several clock cycles to propagate across such long wires causing significant performance loss.

The Multiscalar architecture proposes to engineer a large window and a wide processor at high clock speed by splitting the large window into many small windows and the wide processor into many narrow processing units. Critical path circuits of the processing units may be clocked fast because of their small window size and narrow processing unit width. By localizing critical path wires to the compact processing unit, wire delay problems are mitigated because latency of wire propagation is incurred only on communication among multiple processing units and not on operations within each processing unit. Due to circuit speed constraints, the DEC Alpha 21264 uses a distributed organization for the back-end of the pipeline by grouping functional units into two clusters [42]. The Multiscalar architecture, which predates the DEC Alpha 21264, uses a distributed organization for the entire pipeline, from the fetch stage to the commit stage.

1.2 Multiscalar approach

The Multiscalar architecture employs multiple points of search in the dynamic instruction stream instead of one point of search used by the superscalar architecture. A processing unit establishes a small window at each point of search and searches for independent instructions within its small window. Together, the processing units execute multiple instructions simultaneously exploiting ILP. Figure 1-2 illustrates extraction of ILP in the Multiscalar architecture. Identifying the points of search is an important problem because they determine the degree of independence among the small windows; dense dependences among the small windows result in large amounts of communication, incurring performance loss.

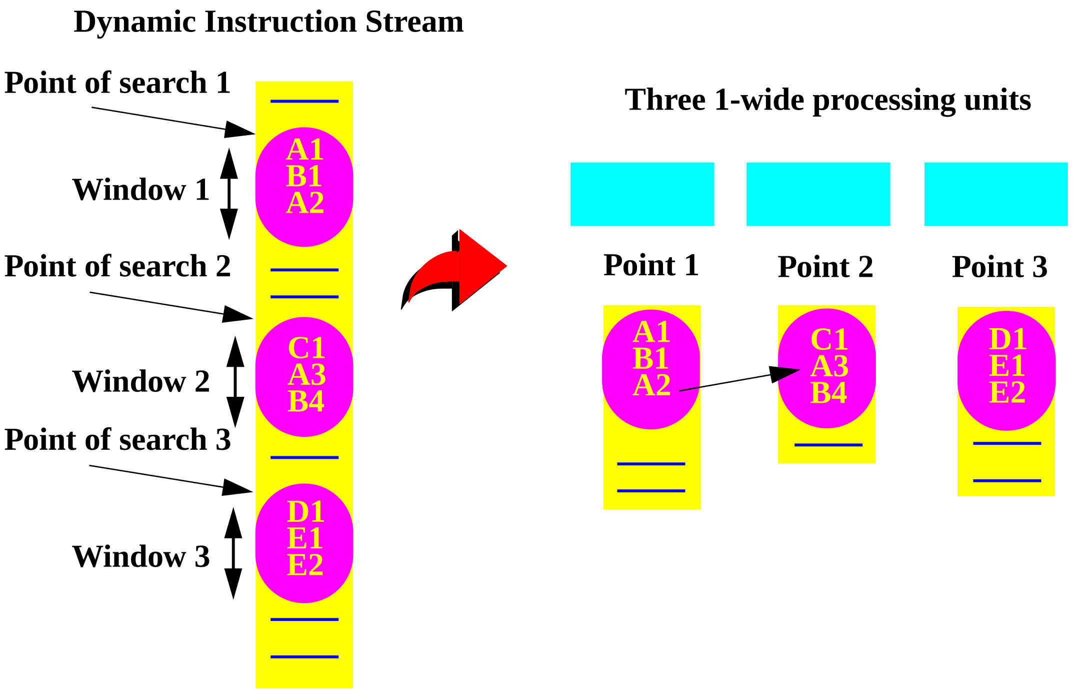


Figure 1-2: Extraction of ILP in the Multiscalar architecture. There are three windows established at three points of search. nstructions in the dynamic stream are named by an alphabet followed by a number; instructions whose names start with the same alphabet form a dependence chain. Window1 contains instructions A1, B1, and A2. Window2 contains instructions C1, A3, and B4. Window3 contains instructions D1, E1, and E2. A3 is dependent on A2. T he three points of search are assigned to three 1-wide processing units for simultaneous execution. Dependence from A2 to A3 is handled to maintain program semantics.

To map the instructions of a sequential program on the distributed processing units of the Multiscalar architecture, the program is partitioned into code fragments called **tasks** which are assigned to the processing units for execution. While threads of parallel or multi-threaded applications use private call stacks, tasks of a program share the global stack, register space, and memory space. Tasks correspond to the points of search for the processing units. Figure 1-3 illustrates how sequential programs are mapped on the processing units of the Multiscalar architecture. To maintain sequential program semantics, inter-task register dependences, inter-task control flow dependences, and inter-task memory dependences have to be honored. Inter-task control flow is handled by speculation and

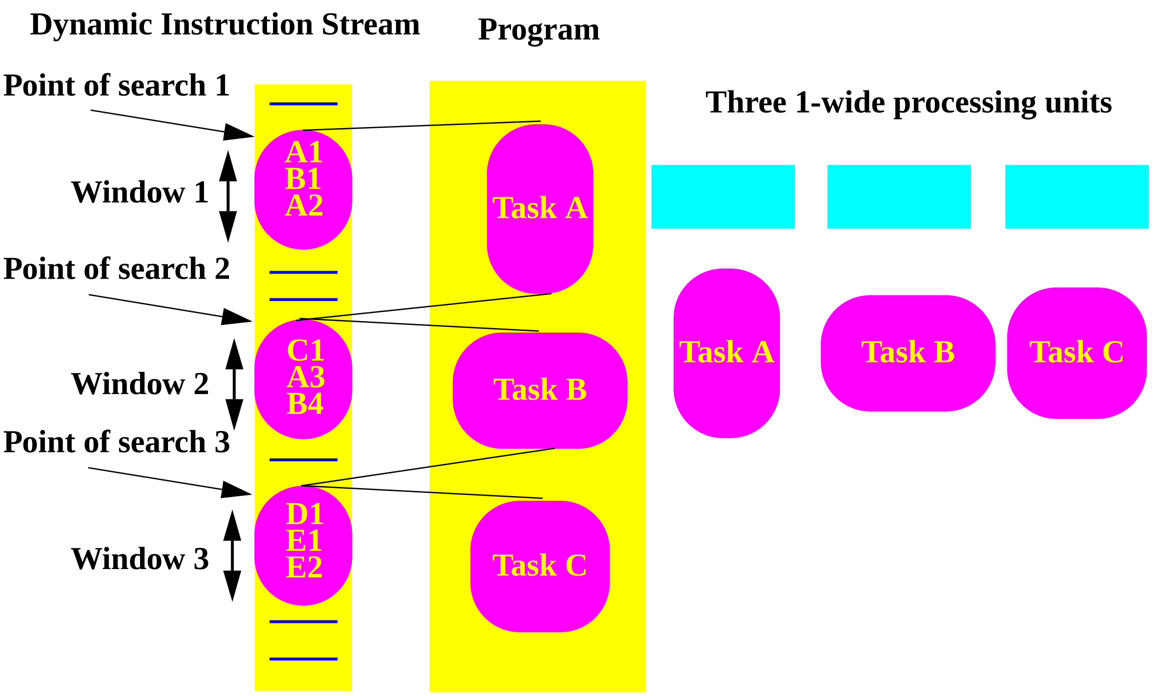


Figure 1-3: Mapping a sequential program on the Multiscalar architecture. the part of the program corresponding to the portion of the dynamic instruction stream shown is partitioned into three tasks, Task A, Task B, and Task C. The three tasks correspond to the three points of search. The three tasks are assigned to three 1-wide processing units for simultaneous execution. Each processing unit establishes its small window of instructions belonging to the task assigned to it.

verification, similar to branch prediction. Inter-task data dependences are dealt with by two sets of mechanisms: register dependences are honored by communication and synchronization and memory dependences are honored by speculation and verification. Intra-task dependences are handled by the processing units, similar to superscalar processors.

In the Multiscalar architecture implementation that I investigate, the compiler partitions sequential programs into tasks. Inter-task register dependences are specified by the compiler, inter-task control flow dependences are specified by the compiler and inter-task memory dependences are handled entirely by the hardware. For register dependences, the hardware activates communication of register values as per compiler specification. For control flow, the hardware speculates on which dynamic task follows the tasks in execution so that it may assign the speculated task for execution without delay; speculation and its verification are based on the information provided by the compiler. Memory accesses from the tasks in execution proceed in arbitrary order with respect to each other speculating that loads and stores from different tasks do not access the same address. Hardware tracks the memory accesses to detect any violation of memory dependences of the application program and performs roll back on misspeculations. In the current implementation, the compiler does not provide any memory dependence information to the hardware.

1.3 Thesis contributions

The physically distributed organization of the processing units and the task-level speculation in hardware combined with imperative, sequential programming model make the Multiscalar architecture sufficiently different from existing microarchitectures. The goals of this thesis are to understand the impact of the fundamental interactions between sequential programs and the novel features of the Multiscalar architecture on overall performance and to identify the key issues involved in the problem of compiling for the Multiscalar architecture.

To make the investigation more concrete, I have devised, implemented, experimented with, and analyzed a few compiler optimizations. I have constructed a compiler to explore optimization opportunities in the context of the Multiscalar architecture, rather than to propose the best solution for a problem addressed by the compiler. Specifically, I have studied task selection, inter-task control flow and register dependence information specification, and inter-task register communication generation and scheduling. I have implemented these optimizations in a widely-used compiler (*gcc*) and measured their impact on the SPEC95 benchmark suite. The compiler techniques apply well-known, conventional compiler analyses (e.g. dataflow analysis) in the new context of the Multiscalar architecture.

1.3.1 Task selection

The Multiscalar architecture requires ordinary, sequential programs to be partitioned into sequential (and not necessarily independent) tasks. From the standpoint of correctness, the architecture does not impose any constraints on the composition of tasks, in terms of control flow or data dependences. At the same time, the choice of tasks is pivotal to achieving high performance. Since existing microarchitectures do not require partitioning of sequential programs into tasks, this problem has not been studied previously in its generality. Automatic parallelization techniques usually partition loops, but not any other control flow structure. In the presence of ambiguous memory dependences, traditional parallelization techniques, not supported by any hardware speculation, are restricted to the options of either completely parallel or entirely sequential execution of loop iterations. Hardware support for speculation in the Multiscalar architecture creates new opportunities for the compiler.

I have identified principal factors that affect performance to include inter-task control flow dependences, inter-task data dependences and task size. I have devised and implemented compiler heuristics that take these factors into account while partitioning programs. The heuristics use traditional graph traversal and dataflow analysis techniques. Although the heuristics handle register dependences and memory dependences alike, the current implementation of the compiler front-end does not provide memory dependence information.

1.3.2 Inter-task dependence information

The Multiscalar architecture uses information about control flow and data dependences present in applications to improve performance. I have devised and implemented a simple compiler scheme to collect inter-task register dependence and inter-task control flow information and convey the information to the hardware in a compact form. The compiler analyses involved in collecting the information are traditional but the information is conveyed and used differently than conventional architectures. For example, register allocators for superscalar architecture perform similar register dependence analysis but do not explicitly convey the information to the hardware for communication purposes.

1.3.3 Register communication

Overlapping register communication with computation is of paramount importance to alleviate performance loss. I have developed several compiler strategies to study the effect of register communication delay on overall performance. These strategies vary the degree of overlap of communication with computation by varying the aggressiveness of the analyses. All of these strategies vary communication timing but do not transform the computation that generates the values involved in communication. Extending this analysis further, I have devised and implemented a static scheduler that moves computation to hide inter-task register communication delay further. Although the scheduler uses traditional transformations to perform code motion similar to superscalar schedulers, the objective, the cost model and the trade-offs of the scheduling process are different from those of superscalar schedulers. Similar to task selection heuristics, the scheduler may be used to hide inter-task memory dependence delay but the current implementation of the front-end does not provide sufficient information. In addition to these delay reducing techniques, I have implemented traditional dead register analysis to reduce register communication bandwidth demand.

1.4 Thesis roadmap

I describe the Multiscalar architecture in more detail, including the execution model, the hardware, the interface between the hardware and the compiler, and performance issues in Chapter 2. At the end of the chapter, I present an overview of the compiler. I discuss task selection in Chapter 3. I identify key task characteristics that impact performance and describe compiler heuristics to partition programs into tasks with suitable characteristics and their implementation. In Chapter 4, I explain the register communication model and describe compiler strategies for register communication generation. I also describe register communication scheduling. In Chapter 5, I present experimental evaluation of the techniques described before. I present measurements of task characteristics, effectiveness of task selection heuristics on overall performance, impact of register communication strategies and scheduling on overall performance. In Chapter 6, I summarize the thesis and draw some conclusions.

Chapter 2

Role of the Hardware and the Compiler

As discussed in Chapter 1, window size and issue-width of superscalar processors cannot be scaled up easily to exploit more ILP without adversely affecting processor clock speed. The need to clock processors fast while exploiting more parallelism has driven architects to investigate distributed hardware organizations. The Multiscalar architecture employs distributed hardware to get the combined effect of a large window of dynamic instructions and a wide-issue processor by splitting the large window into many small windows and the wide-issue processor into many narrow-issue processing units. In this chapter, I start with some basic concepts of the architecture (Section 2.1), then I describe the execution model (Section 2.3), the binary interface between the hardware and the application software (Section 2.4), the hardware (Section 2.5), performance issues (Section 2.6), and the problem of compiling for the Multiscalar architecture (Section 2.7).

2.1 Basic concepts of the Multiscalar architecture

Instead of establishing a large window at a single point of search, the Multiscalar architecture establishes multiple small windows, each at a different point of search. Each window is mapped onto a narrow-issue PU, which executes the instructions in the window. Figure 2-1 illustrates the abstraction of the Multiscalar architecture. Figure 2-1(a) shows a static program partitioned into three tasks and the correspondence between the points of search and the windows in the dynamic stream and the static tasks. The instructions

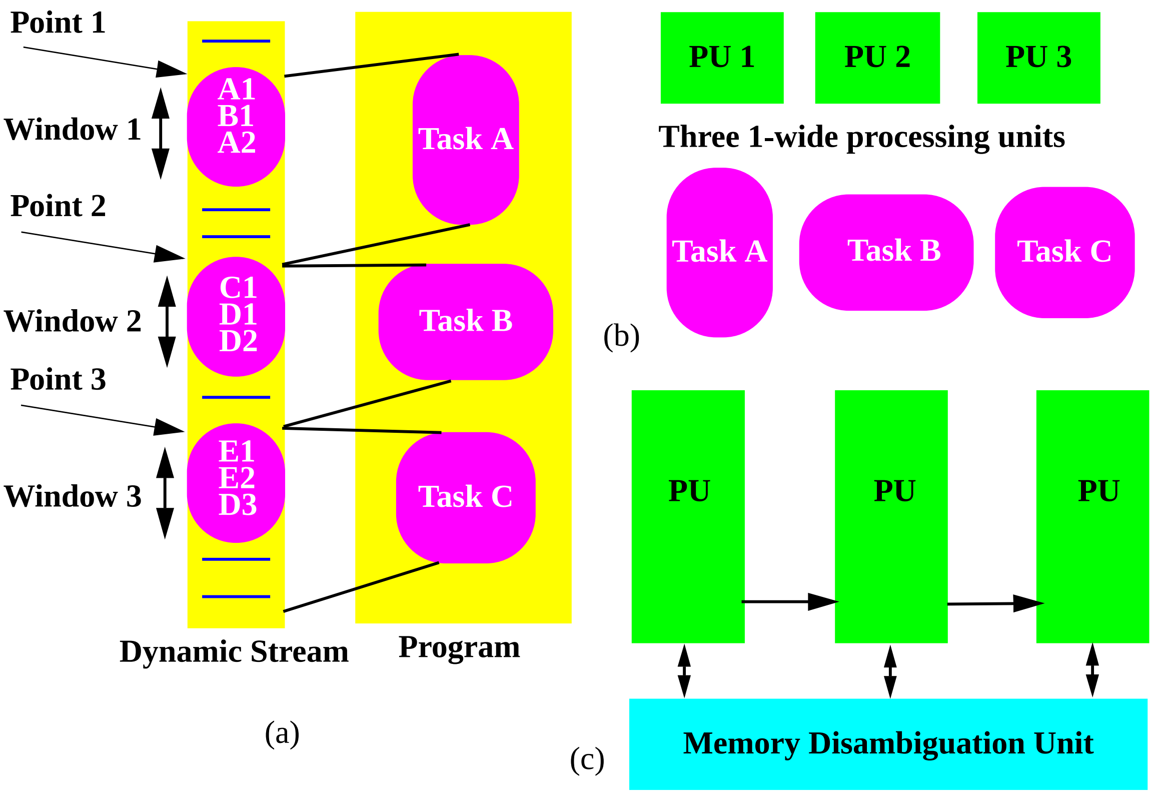


Figure 2-1: Abstraction of the Multiscalar architecture. (a) A static program partitioned into three tasks and three points of search in the dynamic stream with three corresponding windows. The windows at the three points of search in the dynamic stream correspond to the three static tasks. Instructions in the dynamic stream are named by an alphabet followed by a number; instructions whose names start with the same alphabet form a dependence chain. (b) The tasks assigned for execution on processing units. (c) The organization of the hardware. The PUs are connected for communicating data values and the memory disambiguation unit detects dependence misspeculations.

between search points 1 and 2 and search points 2 and 3 correspond to tasks A and B, respectively. Task C corresponds to instructions between search point 3 and the next search point in the dynamic stream. Figure 2-1(b) shows tasks A,B, and C assigned for execution on processing units PU 1, PU 2, and PU 3, respectively. Figure 2-1(c) shows the organization of the hardware for a three PU configuration. The PUs are connected for communicating values and the memory disambiguation unit detects dependence misspeculations.

If the multiple points of search are chosen so that the instructions in one window are mostly independent of the instructions in the other windows, then the PUs can execute in parallel, independent of each other, resulting in high performance. On the other hand, if there is an instruction in one window which is dependent on another instruction in another window, then the dependence is satisfied by communication of values from the producer instruction to the consumer instruction. The consumer instruction waits for the producer instruction to execute and then communicate the required value. Alternatively, an instruction may speculate that it does not depend on any other computation and use the value bound to its register or memory location operand at the time of its execution. If the instruction is dependent on another instruction (i.e., the instruction should have obtained its operand from the other instruction) then a dependence misspeculation is flagged and execution of the consumer instruction (and dependent instructions[1]) is rolled back and restarted. Such dependences result in significant performance loss due to wasted cycles spent either waiting for values or squashing misspeculated computation. Continuing with the example, Task A includes instructions A1, B1 and A2, Task B includes instructions C1, D1 and D2, and Task C includes E1, E2 and D3, as shown in Figure 2-1(a). In this example, A2, D2, and D3 are dependent on A1, D1 and D2, respectively. Since Task B is not dependent on Task A, PU 2 proceeds in parallel with PU 1. Due to the dependence from D2 to D3, Task C depends on Task B causing PU 3 either to stall until PU 2 sends the required value for

[1] In a real implementation, it may be difficult to isolate dependent instructions; in which case all instructions following the misspeculated instruction are rolled back and restarted.

D3 or to misspeculate on the dependence from D2 to D3 dependence and eventually to squash. Therefore, the choice of the points of search is fundamental to the amount of parallelism exploited by the Multiscalar architecture.

2.2 Multiscalar tasks

The compiler identifies the points of search by partitioning application programs into tasks. Each task consists of an entry point, instructions corresponding to the fragment of the application code, and some exit points. Tasks of a program share register space, stack and memory space, whereas threads of a parallel or multi-threaded application use private stacks. By choosing the tasks, the compiler determines the points of search during execution and is responsible for the amount of parallelism exploited by the processor. A task corresponds to a contiguous segment of the dynamic instruction stream, starting from one point of search and ending at the next point of search in the dynamic stream and is executed entirely on one PU. The window established by the PU spans the instructions of the task. The amount of parallelism among dynamic tasks translates to parallelism found in the distributed windows of the Multiscalar architecture. For example, in Figure 2-1(b), instruction D3 in task C is dependent on instruction D2 in task B, causing PU 3 to wait until PU 2 produces and sends the required value. The delay incurred by PU 3 reduces the amount of parallelism exploited and degrades overall performance. Thus, dependences from one task to another result in performance loss. On the other hand, if the two tasks are independent, then the two PUs execute simultaneously and achieve high performance.

2.3 execution model for the Multiscalar architecture

Tasks, obtained by partitioning sequential programs, are assigned to PUs for execution. Each PU executes the instructions of its task to completion. Simultaneous execution of multiple tasks on multiple PUs results in the completion of multiple instructions per cycle. The architecture ensures that the individual execution of each task as well as the aggregate execution of all tasks maintain the appearance of sequential program order.

2.3.1 Control flow

Dynamic prediction unravels the control flow among tasks and each predicted task is assigned to a PU for execution. Execution proceeds by assigning tasks to PUs. After assigning a task for execution, a **control flow speculation** [37] [38] [55] [81] [95] is made which predicts one of the many possible successors of the task, similar to branch prediction employed by superscalar processors [56] [90] [114] [115]. Since the tasks are derived from a sequential program and are predicted in the original program order (similar to branch prediction), the total order among the tasks is unambiguously maintained. Upon completion, tasks are retired in program order to maintain sequential semantics. If control flow speculation is incorrect, i.e., one of the tasks assigned for execution was incorrect, then the incorrect task (and all dependent tasks[2]) is **squashed** similar to a branch misprediction in superscalar processors. A combination of hardware and software mechanisms are used to ensure that control and data dependences are honored as per the original sequential program specification, regardless of what transpires in the actual parallel execution. Figure 2-2 illustrates control flow speculation in the Multiscalar architecture. Figure 2-2(a) shows a part of an application program partitioned into three tasks A, B, and C. The arrows represent inter-task control flow. Task A branches either to task B or elsewhere. Task B branches either to itself of to task C. Figure 2-2(b) shows the tasks execute on a three PU configuration. From the successors of task A, task B is predicted and assigned to PU 2, and from the successors of task B, another invocation of task B is assigned to PU 3. Assuming the second prediction is incorrect, task B executing on PU 3 is squashed and task C is started on PU 3.

[2] In a real implementation, it may be difficult to isolate dependent tasks, similar to isolating independent instructions, as mentioned before; in which case all tasks following the misspeculated task are rolled back and restarted.

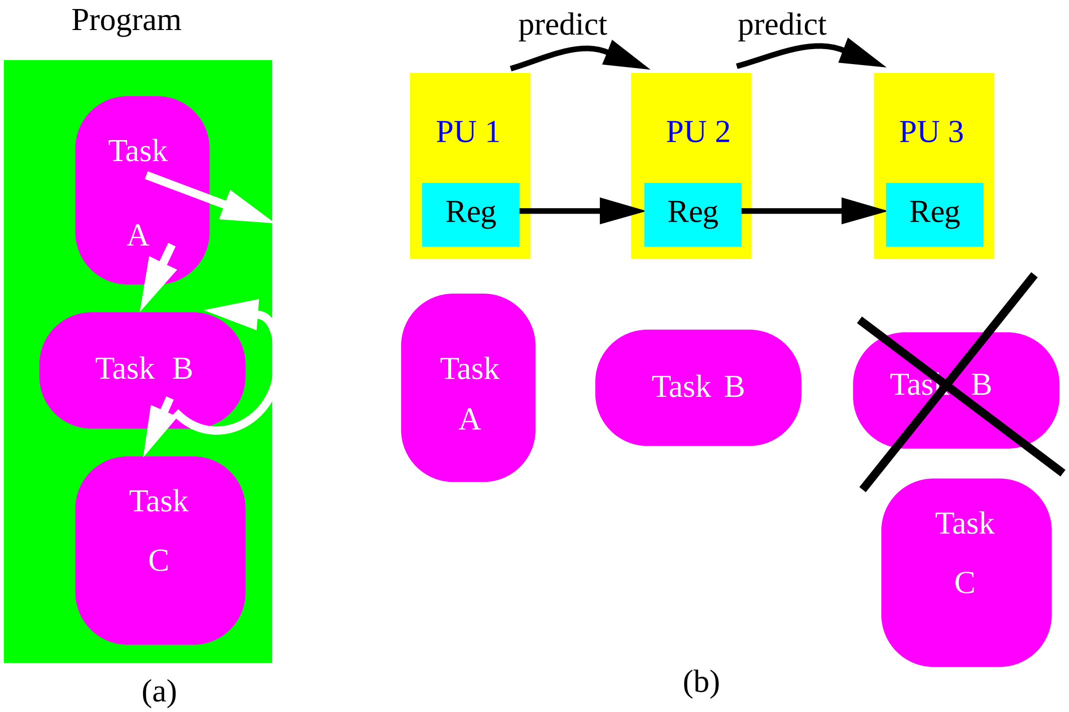


Figure 2-2: Control flow speculation in the Multiscalar architecture. (a) A part of an application program partitioned into three tasks. The arrows represent inter-task control flow. (b) Inter-task control flow speculation. The tasks execute on a three PU configuration. Task B is predicted on PU 2 and another invocation of Task B is predicted on PU 3. The second prediction is incorrect, so task B executing on PU 3 is squashed and task C is started.

2.3.2 Data values

As instructions in tasks execute, data values are produced and consumed within the same task and among different tasks, corresponding to intra-task and inter-task communication, respectively. These data values are bound to memory and register storage locations, similar to superscalar processors. Due to the difference in the size and the nature of register space and memory space, register values and memory values are handled differently. Many ISAs use 32 integer registers and 32 floating point registers, whereas many ISAs support 32-bit (or 64-bit) virtual addresses leading to 2^{32} (or 2^{64}) memory locations.

Register space can be specified by simple bit-vectors since it is small, whereas the large size of memory space makes compact specification harder. In the case of register storage, it is straightforward to identify producers and consumers since all register storage names are known statically via register specifiers. On the other hand, in the case of memory storage, it is difficult to determine precisely the producers and consumers of data values since memory storage names are determined dynamically via address calculations. In the example shown in Figure 2-3, the loop index variable *i* is allocated a register by the compiler.

```
for(i = 0; i < n; i++) {                  Loop:


                                                  ld      rtmp, 0[rptr1]
                                                  beq     rtmp, ri, Not_Eq


        if (*ptr1 == i) {                 Not_Eq:
                                                  add     rtmp, ri, 1
                                                  st      rtmp, 0[rptr2]
        }                                 Continue:
        *ptr2 = i+1;                              add     ri, ri, 1
}                                                 blt     ri, rn, Loop
                  (a)                     Out:
```

Figure 2-3: Register dependences and memory dependences. (a) Source code of a simple loop. *Variables ptr1* and *ptr2* are pointers to unknown addresses. (b) Assembly code for the loop. Registers *rptr1*, *rptr2*, and *ri* hold the values of *ptr1*, *ptr2*, and *i*, respectively. The dependence between the load and the store instructions is ambiguous since the addresses that they are accessing is unknown at compile-time. All the dependences through the register *ri* are unambiguously known since they are directly specified.

Since the addresses of the load and the store instructions are not known at compile time, it is not known whether the load and the store instructions access the same address or not.

But all the dependences involving the register *ri*, which holds the variable *i*, are unambiguously known. Consequently, memory dependences are typically honored via dynamic speculation and verification by the hardware [37] [38] and register dependences are typically honored via synchronization and communication [18] [39] [95], as specified by the compiler. It is important to note that if a memory dependence is known at compile-time then the dependence need not be speculated on, but may be honored via synchronization and communication, like register dependences. Similarly, it may be advantageous to employ speculation on a register dependence, instead of synchronization and communication. Within the spectrum of all data dependences being synchronized and communicated or all data dependences being speculated and verified, any combination of synchronization and communication or speculation and verification of data dependences is possible. Figure 2-4 illustrates register communication in the Multiscalar architecture. Figure 2-4(a) show a part of an application program partitioned into three tasks A, B, and C. Task A produces the value for register d, used by Task B. Figure 2-4(b) shows register communication from task A to task B. PU 1 forwards register d, when the corresponding instruction in task A is executed. Meanwhile, PU 2 executes task B and waits for register d, when the instruction that uses register d is encountered. Task C executes on PU 3 unhindered.

The predicted task speculatively executes on a PU using its private resources unless it needs values computed by another task executing on a different PU. In the case of inter-task register data dependencies, when the producer task computes the required value, it communicates the value to the consumer task. In the case of inter-task memory data dependencies, **memory dependence speculation** [39] [40] [95] is employed; each task speculates that it does not depend on any other task for memory values and performs loads from the specified addresses. If the speculation is incorrect, (i.e., a previous task performs a store to the same address as a load) then a memory dependence violation is detected and the offending load instruction (and dependent instructions) is **squashed**. To prevent frequent memory dependence misspeculations, a hardware mechanism to perform memory dependence prediction and synchronization, which dynamically synchronizes dependent

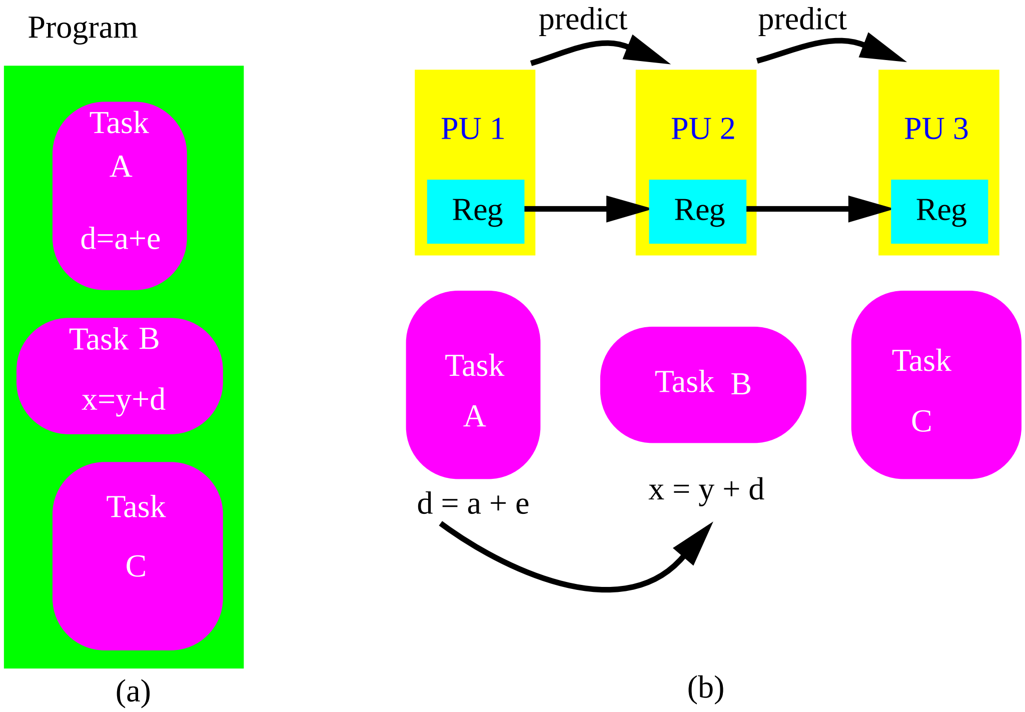


Figure 2-4: Register Communication in the Multiscalar Architecture. (a) A part of an application program partitioned into three tasks. Task A produces the value for register d, used by Task B. (b) Inter-task register communication. PU 1 executes Task A and forwards register d, when the forwarding instruction is executed. Meanwhile, PU 2 executes Task B and waits for register d, when the instruction using the register is encountered. Task C executes on PU 3 unhindered.

loads and stores [73] is employed. In a real implementation, memory dependence squashes and control flow squashes may be handled identically for the purpose of simplifying hardware.

Figure 2-5 illustrates memory disambiguation in the Multiscalar architecture. Figure 2-5(a) shows a part of an application program partitioned into three tasks A, B, and C. Task A stores into address X from which task B loads. Figure 2-5(b) shows detection of violation of memory dependence from task A to task B. PU 2 executes the load in Task B at cycle 1; the load is recorded in the memory disambiguation unit and the current value of

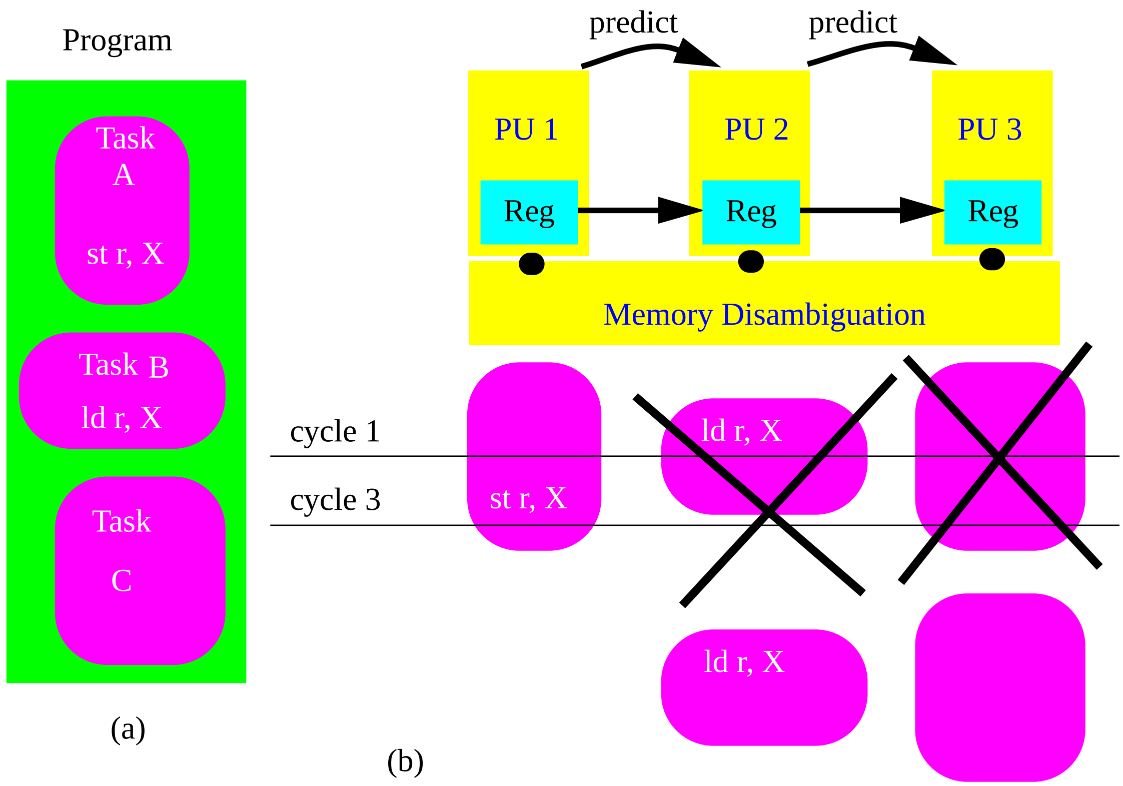


Figure 2-5: Memory disambiguation in the Multiscalar architecture. (a) A part of an application program partitioned into three tasks. Task A stores into address X from which Task B loads. (b) Memory disambiguation. PU 2 executes the load in Task B at cycle 1; the load is recorded in the memory disambiguation unit and the current value of address X is returned. PU 1 executes the store in Task A at cycle 3; the store is also recorded in the memory disambiguation unit. Since a later load executed before a previous store, the memory disambiguation unit detects a memory dependence violation and squashes PU 2 and PU 3. Task B and Task C are restarted on PU 2 and PU 3.

address X is returned. PU 1 executes the store in Task A at cycle 3; the store is also recorded in the memory disambiguation unit. The memory disambiguation unit derives the program order between tasks A and B from the order in which the tasks are predicted and assigned for execution. Since a later load (from the later task B) executed before a previous store (from the earlier task A), the memory disambiguation unit detects a memory dependence violation and squashes PU 2 and PU 3. Task B and Task C are restarted on PU 2 and PU 3.

The amount of computation discarded on a misspeculation (both control flow and data dependence) is an important factor in determining the degree of aggressiveness of the mechanisms (both hardware and compiler) used to perform speculation. Depending upon the granularity of checkpointed state used for rolling back computation on misspeculations, the amount of computation discarded may vary significantly. If entire tasks are rolled back on any misspeculation, then either the accuracy of speculation should be high or speculation should be used infrequently, to avoid rolling back egregious amounts of computation. If only computations that are dependent on misspeculated computation are rolled back, the opportunities lost due to misspeculations may be significantly fewer, obviating techniques to improve speculation accuracy and encouraging frequent speculation. The implementation that I considered to conduct my investigation uses a coarse-grain roll back of entire tasks on control flow misspeculations and data dependence misspeculations.

2.3.3 Sequential semantics

The original program order among the tasks is maintained by keeping track of the order in which tasks are predicted and assigned for execution on the PUs. For the collection of predicted tasks, execution may occur speculatively, but modification of architectural state can only occur non-speculatively. Since tasks execute speculatively, the state (register and memory) produced by it is buffered and cannot update architectural state. When a task completes and all speculations (control flow and memory dependence) have been resolved to be correct, the task is **retired**, i.e., its speculative state is promoted to architectural state. In a simple implementation, tasks are retired in the original program order to maintain sequential semantics; a task is retired only after its predecessor task has been retired. In a more aggressive implementation, if two tasks are completely independent, which may be guaranteed by the compiler, they can be retired out of program order. Although out of program order retiring may increase the complexity of hardware over sequential retiring, performance benefits may be accrued by relaxing the retiring order.

2.4 Application binary interface

The discussion of the execution model leads to the question of the information that the hardware requires to execute a program. The information passed from the compiler to the hardware to execute a program on a conventional superscalar architecture typically includes instructions and static data. Multiscalar hardware requires additional information about the tasks of the program.

When a task is assigned to a PU for execution, the hardware needs the start address of the next task to assign it to another PU for execution. The PU executing a task needs to know when the task ends. The start and end information of tasks are collectively called as inter-task control flow information. Apart from task boundaries, if tasks employ synchronization and communication to enforce inter-task data dependences (registers and memory) then the hardware requires specification of any such synchronization and communication between the producer task and the consumer task involved in the data dependence. The PU that executes the consumer task needs to know which values are to be obtained from other tasks and when the values are available. The PU that executes the producer task needs to know which values are to be sent to other tasks and when the values may be sent. The information of when a data value may be sent from a task and which data values will be sent by other tasks is collectively called as inter-task data dependence information.

The additional information about the tasks may be passed on from the compiler to the hardware via the program binary. Alternatively, some auxiliary hardware may garner the information from conventional program binary. Since I wanted to explore compilation techniques, I opted to use the compiler to obtain and convey such information to hardware. In summary, program binaries include: (1) the instructions of the program, (2) inter-task control flow information, and (3) inter-task data dependence information. The instructions of the program remain the same as a regular superscalar, in the implementation I consider.

2.4.1 Inter-task control flow information

When a task is assigned for execution, the hardware predicts one of the successors of the task to be assigned for execution. The hardware needs to know the possible successors of the task; not predicting the probable successor task until its identity is known incurs performance loss. The compiler explicitly provides a list of successors of the task called **targets** and the hardware predicts one of the targets [95]. It is important to note that if the compiler does not provide all of the targets of a task, the hardware could learn and remember the unspecified targets as they are encountered in execution. The advantage of not specifying a target is smaller binary (and instruction memory resources) and the disadvantage is that the first time an unspecified target is encountered, the next task cannot be assigned until the target is computed, which may lead to performance loss. Apart from the successors of a task, the PU needs to know where a task ends, so that it may terminate execution. The compiler tags the instructions at the boundaries of tasks with a few extra bits called the **task-exit bits**. When the PU encounters an instruction with the task-exit bits set, it stops execution and the task ends. Branch instructions lead to two possible control flow paths and the task-exit bits identify the paths that exit the task.

2.4.2 Inter-task data dependence information

I discuss register dependences first and then memory dependences. If inter-task register dependences are to be honored by the compiler, then the compiler needs to specify explicit synchronization and communication among dependent instructions. In order to keep the discussion simple, let us assume that all inter-task register dependences are enforced by the compiler via synchronization and communication. In a real implementation, some of the dependences may be honored by the hardware via speculation and verification and the rest may be handled by the compiler.

The compiler determines the set of register values that may be produced by a task and the set of register values that may be consumed by the task. If a task consumes a data value produced by another task, then the consumer task waits until the value is sent to it. In accordance with sequential semantics, the last update of a register in a task should be **forwarded** [38] [95] to succeeding tasks[3]. If a register is guaranteed not to be modified by a task, then it may be propagated to successor tasks as and when it arrives. Since the PU cannot determine a priori which instructions comprise its assigned task (the instructions may not even have been fetched), it cannot know (1) which registers are guaranteed not to be modified, and (2) which instruction performs the last update to a register that needs to be forwarded. Waiting until all instructions in a task have executed so that all registers are updated serializes the execution of tasks, incurring performance loss. The compiler provides the set of registers which may be modified as a bit-vector called the **create mask**. The compiler determines the last update of each register and tags the instruction with a few extra bits called the **forward bits**, similar to task-exit bits. When a PU encounters an instruction with forward bits set, called a **forwarding instruction**, it forwards the value corresponding to the destination register of the instruction. If a register is forwarded in one control flow path but there are no instructions that modify the register in another path, an extra instruction called a **release instruction** may have to be inserted to send register values. The targets and the create mask of a task are collected together in its **task descriptor**.

If memory dependences are handled entirely by the hardware, the compiler need not provide any information. As explained before, since it is difficult for the compiler to determine and compactly specify memory dependences, hardware is employed to enforce memory dependences.

[3] If speculation and verification is employed, multiple values for a register may be sent. In that case, the value corresponding to the last update of the register should be sent last or tagged specially, to maintain sequential semantics.

2.4.3 An example of a Multiscalar program

In order to clarify the kind of information required, I present an example of a code snippet and illustrate its execution. Figure 2-6 shows the source code and the assembly code with information specific to Multiscalar tasks.

The example code in Figure 2-6(a) is a part of a simplified cache simulator; it contains a loop which scans down the array *trace*. *Trace* holds the addresses of the simulated memory accesses and the array *cache* simulates the cache. *TAG_MASK* and *IDX_MASK* are bit masks used to select the cache tag bits and cache index bits, respectively. *HEAP_MASK* is a bit mask used to identify if an address falls into the heap section of the simulated virtual memory space. The variables *tag* and *idx* hold the cache tag bits and cache index bits, respectively, of each memory address. *Heap_access* keeps count of the number of memory addresses that fall into the heap section. *Cache_hit* keeps count of the memory accesses that hit in the cache, i.e., those accesses for which the index maps to a valid cache entry and the tag matches the tag stored in the cache entry. For accesses that do not hit in the cache, the cache entry corresponding to the current index is updated with the current tag.

The corresponding assembly code in Figure 2-6(b) shows variables *i, n, address, idx, tag, heap_access,* and *cache_hit* in registers *ri, rn, radd, ridx, rtag, rheap,* and *rhit*, respectively. Registers *rtrace* and *rcache* hold the address of the cells of *trace* and the base address of *cache,* respectively.

Let us assume that the loop body is partitioned into two tasks, shown by shaded regions in Figure 2-6(b): one from the label *Loop* to the add instruction above the label *Not_Heap* and the other from the label *Not_Heap* to the label *Continue* (the jump instruction at end of the loop body). There are many other task partitions possible but the partition chosen here serves to illustrate the kind of information required to execute tasks.

```
while (i++ < n) {
        address = trace[i];
        idx = address & IDX_MASK;
        tag = address & TAG_MASK;
        if (address & HEAP_MASK)
             heap_access++;
        if (VALID(cache[idx]) &&
         (tag == TAG(cache[idx]))
             cache_hit++;
        else
             TAG(cache[idx]) = tag;
}
```

(a)

```
Loop:
        add     ri, ri, 1
        add     rtrace, rtrace, 4
        bge     ri, rn, Out
        ld      radd, 0[rtrace]
        andi    ridx, radd, IDX_MASK
        andi    rtag, radd, TAG_MASK
        andi    rtmp, radd, HEAP_MASK
        beq     rtmp, r0, Not_Heap
        add     rheap, rheap, 1

Not_Heap:
        add     rblock, rcache, ridx
        ld      rvalid, VALID[rblock]
        beq     rvalid, r0, Cache_Miss
        ld      rtmp, TAG[rblock]
        bne     rtag, tmp, Cache_Miss
        add     rhit, rhit, 1
        jmp     Continue
Cache_Miss:
        st      rtag, TAG[rblock]
Continue:
        jmp     Loop
Out:
```

(b)

Targets: Task2, Task3
Create Mask: ri, rtrace, radd, ridx, rtag, rtmp, rheap

```
Task1:
F       add     ri, ri, 1
F       add     rtrace, rtrace, 4
ST      bge     ri, rn, Task3
F       ld      radd, 0[rtrace]
F       andi    ridx, radd, IDX_MASK
F       andi    rtag, radd, TAG_MASK
F       andi    rtmp, radd, HEAP_MASK
ST      beq     rtmp, r0, Not_Heap
F S     add     rheap, rheap, 1
Not_Heap:
```

(c)

Targets: Task1
Create Mask: rblock, rvalid, rtmp, rhit

```
Task2:
F       add     rblock, rcache, ridx
F       ld      rvalid, VALID[rblock]
        beq     rvalid, r0, Cache_Miss
F       ld      rtmp, TAG[rblock]
        bne     rtag, tmp, Cache_Miss
F       add     rhit, rhit, 1
        jmp     Continue
Cache_Miss:
        release rhit, rtmp
        st      rtag, TAG[rblock]
Continue:
S       jmp     Task1
Task3:
```

(d)

Figure 2-6: An example of a Multiscalar program. (a) The source code of a loop of a simplified cache simulator. (b) The assembly code for the loop. The loop body is partitioned into two tasks. (c) and (d) The two tasks and their descriptors. **F**, **ST**, and **S** denote forward register, exit if taken, and exit always, respectively.

In Figure 2-6(c), the two branch instructions are annotated with task-exit taken bits (indicated by ST) causing the hardware to terminate the task if either of the branches are taken. The add instruction at the end also has its task-exit bits set (indicated by S) causing the task to terminate. In Figure 2-6(d), the jump instruction has its task-exit bit set (indicated by S) causing the task to terminate at the end of the loop.

In Figure 2-6(c), since all the integer instructions and the load instruction of Task1 are last updates of the respective destination registers, they have the forward bits set (indicated by F). Any register on the create mask that was not forwarded during the execution of the task is forwarded after the task terminates. For example, if the second branch of Task1 is taken then *rheap* is not forwarded by any instruction and it is automatically forwarded at the end. Task2 is similar to Task1 except for the release instruction which forwards the registers *rhit* and *rtmp*. If either of the first two branches of Task2 are taken, then the registers *rhit* and/or *rtmp* are not forwarded by any instruction; instead of letting the hardware forward them at the end (similar to *rheap* of Task1), the release instruction forwards them earlier because subsequent tasks may be waiting for them.

To explain how load and store instructions are executed, let us assume that two iterations of the loop in Figure 2-6(a) are executed on a four-PU configuration. Task1 and Task2 of iteration 1 and Task1 and Task2 of iteration 2 are executed on PU1, PU2, PU3, and PU4, respectively. PU1 forwards registers *ri*, *rtrace*, *radd*, *ridx*, *rtag*, and *rheap* to PU2 when the corresponding forwarding instructions are executed. In addition to forwarding registers *rblock*, *rvalid*, and *rhit*, PU2 also propagates the registers received from PU1 to PU3. PU3 consumes the registers received from PU2 and forwards new values and propagates old values for the corresponding registers. It is possible that iterations 1 and 2 index into the same cache entry. If iteration 1 simulates a cache miss and iteration 2 simulates a cache hit, the store instruction of PU2 (Task2, iteration 1) and the load instruction of PU4 (Task2, iteration 2) access the same entry of cache. Depending upon the timing of execution, either the store from PU2 reaches the ARB before the load from PU4 or vice-versa. If the store is ahead of the load then the load gets the correct value of the cache entry; otherwise, when the store is executed, the ARB detects a memory dependence violation and squashes PU3 and PU4. Task1 and Task2 of iteration 2 are reexecuted on PU3 and PU4.

2.4.4 Other alternatives to binary interface

There are alternative schemes to provide task information to the hardware other than tagging instruction with extra bits. Assuming that the program binary does not contain any task information, instead of a compiler some auxiliary hardware could scan the instructions as they are fetched and demarcate tasks. Once a task is demarcated, the instructions can be scanned and inter-task control flow information and inter-task data dependence information may be determined. Unlike the compiler which stores the task information in the program binary, the hardware scheme could buffer the information in a special cache or in the instruction cache itself (similar to an instruction cache which contains predecoded instructions). The compiler may be able to produce better tasks than the hardware because it can include more analyses than the hardware. Since I am exploring the compiler, I will not elaborate on this approach any more.

2.5 Hardware implementation of the Multiscalar architecture

All the information required by the hardware is included in the program binary. The Multiscalar hardware executes a program by fetching and executing instructions, similar to a superscalar processor. The task information included in the binary is used by the hardware to coordinate the execution of tasks in the program. The Multiscalar processor is organized as a collection of processing units (PUs), each of which has its own fetch and issue logic, register file, pipelined datapath hardware, and load-store queues, similar to a superscalar processor pipeline core. The PUs are connected to a first level instruction cache which may be backed up by next level caches. The PUs are connected[4] together to

[4] Although logically any interconnect may be used, since data values and other information can be communicated from a predecessor task to a successor task only (and not the other way), a simple unidirectional ring may be easier to engineer.

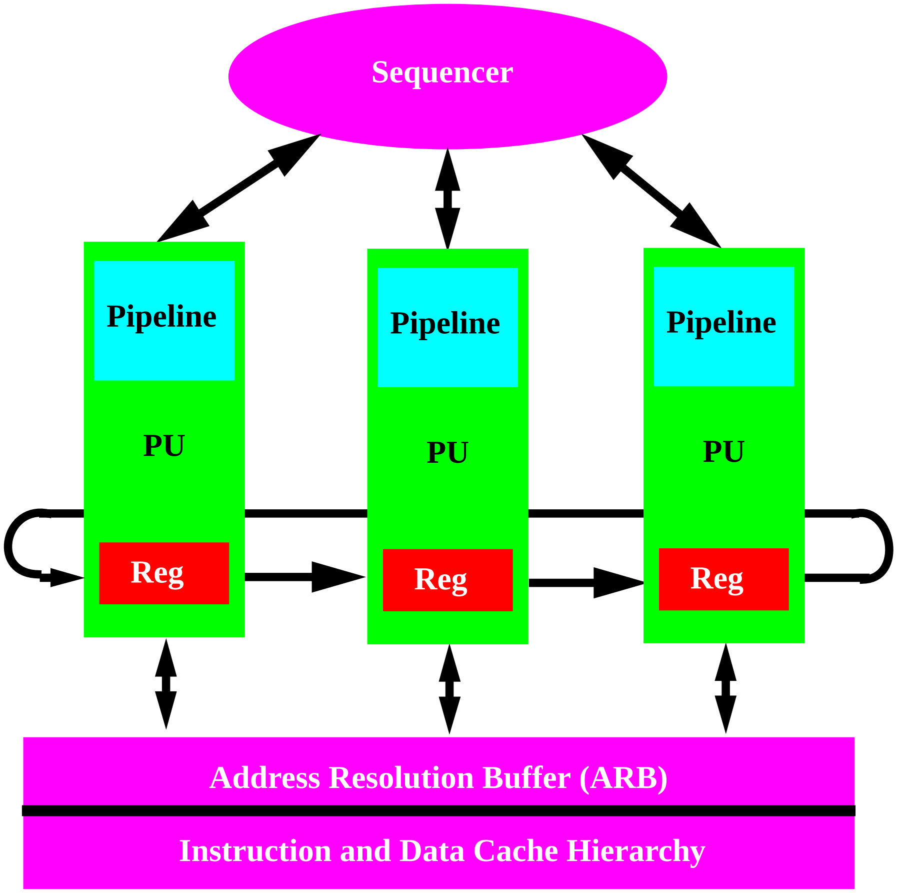

Sequencer
Pipeline
PU
Reg
Pipeline
PU
Reg
Pipeline
PU
Reg
Address Resolution Buffer (ARB)
Instruction and Data Cache Hierarchy

Figure 2-7: Hardware for the Multiscalar architecture. The Sequencer performs control flow speculation and assigns tasks to the PUs for execution. The PUs include narrow-issue pipeline datapaths with fetch, decode, and issue logic as well as a private register file. All memory accesses from the PUs are sent to the ARB to detect memory dependence violations. The ARB is backed up by the rest of the data cache hierarchy. The PUs are also connected to the instruction cache hierarchy.

communicate data values and other information from one unit to another [36] [38]. Apart from the PUs, there is a hardware predictor which predicts the tasks to be executed. The sequencer assigns tasks to PUs in the predicted program order and keeps track of the program order among them. The PUs are also connected to the memory disambiguation mechanism called the address resolution buffer or ARB [40] to handle loads and stores. The ARB enforces the original program order among memory operations performed by different PUs; any load and store to the same address performed out of program order is detected and corrected by the ARB. The ARB also buffers all store values of a task until the task is retired, at which time the ARB updates the first level cache. The ARB is backed up by the first level data cache and the rest of the memory hierarchy, which hold only non-speculative values.

2.5.1 Control flow: Sequencer

The sequencer assigns a task for execution on a PU and fetches the task descriptor of the task, using the starting PC of the task called the **task PC**. Task descriptors are cached in the **task cache** to reduce latency of access. The sequencer then predicts one of the targets specified in the descriptor using task prediction tables similar to branch prediction [37] [38] [55] [90] [95] [114] tables used by superscalar processors. The sequencer probes the prediction tables with the task PC and the prediction tables generate a target number based on past history of the targets taken by the task. The sequencer uses the target number to probe the task descriptor and gets the task PC for the next predicted task and continues task assignment if the next PU is free.

2.5.2 Register data: Unidirectional communication ring

When the sequencer assigns a task to a PU for execution, the PU starts fetching and executing instructions from the task PC. If all the input register operands of an instruction is not available, then the instruction is kept pending until the required value is received from some predecessor task. When a register value is received from the previous PU, the corresponding local register is updated with the value. If the register is not in the create mask of the task (i.e., it is guaranteed not to be modified by the task), then the value is passed on to the next PU; otherwise, the register value is not passed on any further and it will be sent when the corresponding forwarding instruction is encountered. When an instruction with forward bits set is executed, the destination register value is sent on the ring to the next PU.

2.5.3 Memory data: Address resolution buffer

Every load and store from any of the PUs access the ARB. The ARB is organized like a cache and indexed using the memory access address. Every load and store is recorded in the ARB and stores deposit the values in it and do not update the data cache. A store to an address from a task checks for loads from the same address that have already been performed by later tasks. If any such load is found (i.e., a memory dependence violation is said to have occurred) then the task that performed load (and all later tasks) are squashed. When a task retires, its buffered store values are written back to the data cache and the ARB entries are cleared. The data cache holds only architectural state and no speculative state.

2.5.4 Sequential semantics

The hardware maintains the program order among the predicted tasks by assigning them in the order to the PUs in the direction of the unidirectional communication ring. This assignment maps the logically ordered tasks on the physically ordered PUs. In order to continue to maintain this ordering even through squashes, any squash results in the offending task and all logically succeeding tasks (inferred from the order of the PUs to which they are assigned) being squashed. This squash model maintains the one-to-one correspondence between the logical order of the tasks and the physical order of the PUs. In order to maintain correctness of data values, a task is retired only after all register values are sent and all memory values are written back from the ARB to the data cache.

2.6 Performance Aspects

In this section, I add timing information to the functional description of execution of tasks to account for execution time of tasks and understand important performance issues. When a task is assigned for execution, two possibilities arise: (1) the task completes and is retired, or (2) an incorrect speculation (control flow or memory dependence) occurs and the task is squashed.

2.6.1 Scenario 1: Task is retired

When a task is assigned for execution, it starts out by fetching instructions from the start PC and filling the pipeline of the PU with instructions. The time associated with filling the pipeline is classified as **task start overhead**. Each instruction executes as long as its input operands are available, similar to a superscalar processor. If an input operand is not available, then the instruction waits until the value is available. If the instruction waits for a value to be produced and communicated by another task, then the associated wait time is classified as **inter-task data communication delay**. If the instruction waits for a value to be produced by a previous instruction in the same task, then the associated wait time is classified as **intra-task data dependence delay**. As soon as the required value is available, execution proceeds and eventually the task ends. After completion, the task waits to retire; the associated wait time is classified as **load imbalance** because the wait time is incurred due to the difference in the amount of computation performed by the task and its predecessor tasks. When the task is allowed to retire, it commits its speculative state to architectural storage; the associated time is classified as **task end overhead**. After the task retires, the PU is ready to execute another task. Figure 2-8(a) illustrates the various phases of scenario 1.

2.6.2 Scenario 2: Task is squashed

When a task is assigned for execution, it proceeds as explained above until an incorrect speculation is detected. Either the task itself or one of its predecessors is detected to have misspeculated and the task is squashed and a new task is assigned. The entire time since the start of the task, irrespective of whether the task was waiting for values or executing instructions, is classified as **control flow misspeculation penalty** or **memory dependence misspeculation penalty**, as the case may be. Since a misspeculation may cause sev-

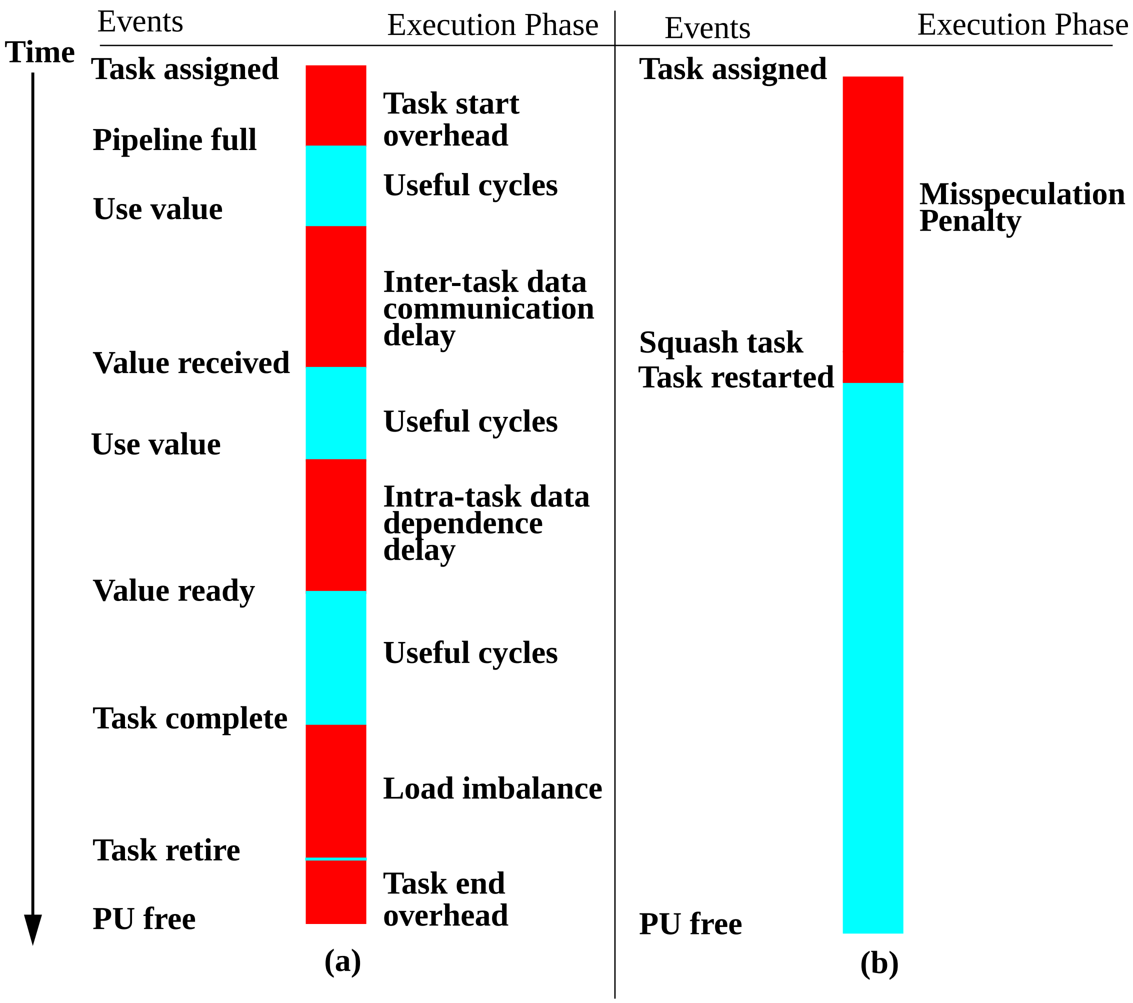


Figure 2-8: Time line of the execution of a task. Time extends downwards. Events occurring during the execution are shown on the left of the timeline. The names for the various time-phases corresponding to the events are shown on the right of the timeline. (a) Scenario 1: Task executes to completion. (b) Scenario 2: Task is squashed and restarted.

eral tasks to be squashed (the offending task and all its successors), the misspeculation is associated with the sum of all the individual penalties of each of the squashed tasks. Figure 2-8(b) illustrates the various phases of scenario 2.

2.6.3 Performance improvements through the compiler

The issues listed above are addressed by specific compiler techniques which mitigate the performance loss associated with each of the categories. Complementary hardware techniques also are employed to alleviate performance problems. Task start overhead and task end overhead are amortized by selecting large tasks. If the start overhead is two cycles to fill the pipeline of the PU, then the compiler may be configured to select tasks which are 20 instructions or more to keep the overhead to less than 10%, assuming the PU may issue at most 1 instruction per cycle. Inter-task data communication delay is tackled by two techniques: (1) Task selection heuristics select tasks with as few inter-task data dependences as possible to avoid inter-task data communication as much as possible and (2) The inter-task data dependences that could not be avoided by the task selection heuristics are handled by scheduling; producers and consumers involved in the inter-task dependences are moved up (early) and down (late) in their respective tasks. The required data value is produced early and communicated before it is required by the consumer. Intra-task dependence delay is handled by traditional instruction scheduling. Load imbalance is alleviated by controlling the variation in task sizes. By imposing an upper limit on the number of instructions included in a task, task selection heuristics may disallow large variations in task sizes. Performance loss due to misspeculation (both control flow and memory dependence) is mitigated by two techniques: (1) Task selection heuristics partition programs into tasks with as few control and memory dependences[5] as possible to avoid inter-task dependences that are misspeculated and (2) The inter-task dependences that could not be avoided by the task selection heuristics are handled by scheduling; verification of the speculated dependences is scheduled early to reduce misspeculation penalty.

[5] Memory dependences are hard to determine at compile-time due to ambiguous pointer accesses and inter-procedural memory accesses. The heuristics described in this thesis (in Chapter 3) use simple analysis of memory dependences involving global scalar variables referenced by name.

2.7 The problem of compiling for the Multiscalar architecture

Compiling for the Multiscalar architecture involves discovering independent computations in a sequential program. But unlike a VLIW, superscalar or a parallelizing compiler, the Multiscalar compiler need not provide absolute guarantees of independence because of the extensive hardware support for speculation[6]. Control speculation and memory dependence speculation in hardware relieves the compiler of the burden of guaranteeing complete independence or correct synchronization among Multiscalar tasks. The flexible granularity of tasks allows the compiler to select tasks on the basis of the parallelism present in the program and does not force the compiler to engineer the code to fit the granularity of the hardware. Superscalar and VLIW compilers have to engineer (via techniques like code motion across basic blocks and software pipelining) the inherent (possibly complex, in terms of control and data dependencies) grain of the program into the rigid n-wide instruction structure, which the architectures execute.

The Multiscalar architecture provides synchronization mechanisms for register dependences, which the compiler can detect. For the cases involving memory dependences, in which the compiler may not be able to perform an accurate analysis, the architecture allows the compiler to aggressively speculate on ambiguous dependences. In conventional architectures, instead, the compiler is forced to generate sequential code. Needless to say, presence of dependence would prevent application programs from executing faster. Nevertheless, aggressive solutions vastly improve performance in situations where there is parallelism which cannot be established with absolute guarantees by the compiler.

In more concrete terms, the Multiscalar compiler identifies computations which are mostly but not necessarily independent and bundles them into tasks. The compiler partitions and schedules around known control flow and data dependences. For unknown and

[6] VLIW and superscalar architectures provide support for speculation via predicated instructions [54] (e.g., conditional move instructions).

ambiguous dependences, the compiler speculates that there are no dependences and partitions programs ignoring such dependences; the compiler attempts to keep the misspeculation penalty low by keeping the size of tasks modest so that misspeculations do not discard large amounts of computations.

The responsibilities of the compiler include partitioning sequential programs into tasks with few dependences among each other, maintaining correctness by specifying control and data dependencies among the tasks to the hardware, and improving performance by streamlining control and data dependences among the tasks. Corresponding to these requirements, the problem of compiling for the Multiscalar architecture comprises four parts: (1) devising heuristics to obtain suitable tasks, (2) specifying inter-task register communication and inter-task control flow to maintain correctness, (3) scheduling inter-task data (both memory and register) communication to mitigate performance loss, and (4) evaluating compiler strategies.

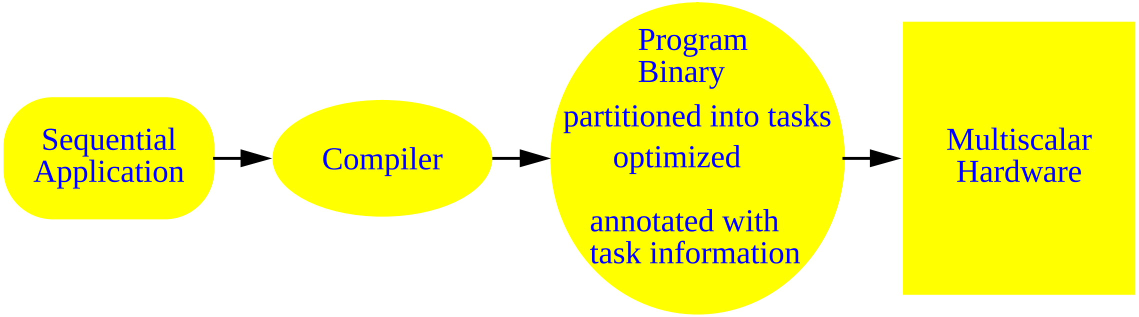


Figure 2-9: Role of the compiler for the Multiscalar architecture. The compiler compiles sequential applications down to the hardware. The compiler partitions the code into tasks and performs several optimizations and generates a program binary which includes task information (register create masks, register forward bits, task-exit bits, and task targets).

Task selection involves determining control and data dependences among instructions and collecting instructions into tasks such that most dependent instructions are included in the same task. Specifying inter-task register communication and inter-task control flow includes determining the last update of a register in a task and identifying the branches that lead control flow from one task to another. Register communication scheduling involves identifying the producer and consumer instructions of inter-task register dependences. The scheduler moves the producer instructions up and the consumer instructions down in their respective tasks so that the time taken to produce the value and communicate it is overlapped with some useful computation in the task of the consumer instruction.

Like other compilers, the Multiscalar compiler faces both correctness and performance issues. Correctness issues include inter-task register dependences and inter-task control flow and performance issues include task selection and register communication scheduling.

2.7.1 Correctness issues

Inter-task register dependences and inter-task control flow are completely specified by the compiler to the hardware. Any instruction that is the last update of a register, irrespective of the control flow paths within the task, must be tagged to forward its destination register[7]. The compiler cannot guarantee sequential semantics if this rule is violated. Control flow path that exits a task must lead to the entry point, and not the middle, of another task. Since register forwarding is set up by the compiler assuming tasks have a single entry point, sequential semantics cannot be guaranteed if this rule is violated. Similar to other instruction scheduling schemes, register communication scheduling also involves correctness issues as well. Code motion employed by register communication scheduling must conform to sequential semantics.

[7] If a register is forwarded multiple times due to hardware or compiler speculation, the last forward must send to the correct value.

2.7.2 Performance issues

Task selection is the most important performance issue of the Multiscalar architecture. Since there is no restriction on the control flow and data dependences that may exist within a task and among tasks, any arbitrary, contiguous sequence of static instructions is a valid task; the choice of tasks is entirely a performance consideration. To expose as much parallelism as is in the program, task selection should result in as few inter-task dependences as possible. Any inter-task dependences result in either tasks waiting for values or tasks speculating that there are no dependences and squashing. If an inter-task dependence is unavoidable, then tasks should be selected such that the producer instruction of the value involved in the dependence is close to the beginning of its task and the consumer instruction of the value is close to the end of its task. Thus, producer instruction is executed early and consumer instruction is executed late resulting in overlap between communication and computation. If task selection does not succeed in achieving this overlap, then code scheduling, which employs static code motion to move the producer up to the beginning of its task and the consumer down to the end of its task, should be done.

2.7.3 Organization of the compiler

The techniques discussed above are implemented as a series of compiler optimizations phases. The organization of the compiler, which is derived from *gcc*, is shown in Figure 2-10. At the top level, gcc parses, optimizes, and compiles down to assembly one input function after another. After a series of traditional phases like jump optimizations, common sub-expression elimination, and loop optimizations, the Multiscalar compiler performs loop optimizations specific to the Multiscalar architecture called loop restructuring. After the program is partitioned into tasks, the compiler schedules register communication. Register allocation is performed after that in the usual manner. At the final code generation phase, the compiler annotates the assembly code with inter-task register communication and control flow information.

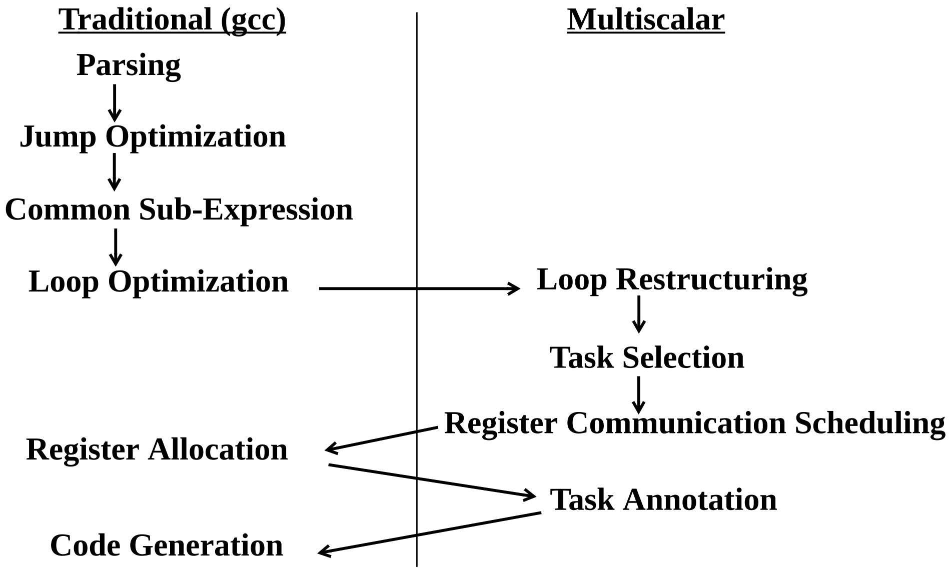


Figure 2-10: Organization of the compiler for the Multiscalar architecture. The different phases of the compiler and the phase ordering implemented in the compiler. The traditional phases of Gcc are listed on the left and the phases specific to Multiscalar are on the right.

Task selection and register communication scheduling take advantage of profile information (basic block frequencies), if available. Since many decisions made by the compiler during task selection and register communication scheduling depend on the relative importance of optimization opportunities, frequency counts obtained from dynamic profiling are used to prioritize the opportunities. Profile information is optional in that the compiler generates Multiscalar binaries even if the information is not available.

Chapter 3

Task Selection

Task selection is one of the most important problems for the compiler; the amount of parallelism exploited by the Multiscalar hardware while executing a program depends significantly on how the program is partitioned into tasks. While a good task selection may result in the program partitioned into completely independent tasks leading to performance improvements linear with respect to the number of processing units, a poor task selection may lead to the program partitioned into dependent tasks resulting in performance worse than that of a single processing unit, due to overheads resulting from distributing hardware resources. For example, if register dependences through a critical path are exposed across tasks, register communication may add extra latency cycles to the critical path. Centralized register value bypassing used in superscalar processors may not incur this overhead[1].

The guiding principle used to select tasks is that control and data dependent computation should be grouped into a task so that synchronization and communication or speculation

[1] This example does not take clock speed differences into consideration.

and verification are minimized. Any inter-task communication or speculation may result in loss of performance due to communication delays or misspeculation penalties. There are two issues in applying the principle: (1) it is hard to determine all the data dependences in a given application due to ambiguous memory dependences (as explained in Section 2.3.2) and (2) even if the dependence information is accurate, data dependences, control dependences, load imbalance, and task overheads (as explained in Section 2.6) often impose conflicting requirements. Sarkar [89] showed that partitioning simple functional programs into tasks to execute on a multiprocessor is NP-Complete. He modeled program execution time as a function of the amount of work done by each task and data communication delay due to inter-task dependences. Even without including considerations about speculation in the model[2], the problem of partitioning programs into tasks for optimal performance is intractable.

I handle the first issue by determining register dependences and a few simple memory dependences involving statically named variable accesses. Any ambiguous memory dependence involving pointers and heap structures are not analyzed and are ignored. Hardware support for memory dependence speculation allows the compiler to ignore ambiguous memory dependences, as explained in Section 2.7. To address the second issue, I have devised a set of heuristics to select tasks for high performance.

This chapter describes the problems involved in partitioning a sequential program into Multiscalar tasks and my solutions to the problems. I define tasks in Section 3.1 and discuss the relationship between performance issues and task characteristics in Section 3.2. In Section 3.3, I discuss important criteria used to select tasks with favorable characteristics. I describe heuristics which incorporate the criteria in Section 3.4, and present some details of my implementation in Section 3.5.

[2] In other words, the assumption that speculation always succeeds would keep the simple model unchanged.

3.1 Definition

A Multiscalar task is defined to be a connected, single-entry subgraph of the static control flow graph (CFG) [1] of a sequential program. A task corresponds to a contiguous fragment of the dynamic instruction stream that may be entered only at the first instruction of the fragment. There are no other constraints on tasks except that they cannot comprise disconnected parts of the dynamic instruction stream. A task may comprise a basic block[3] [1], multiple basic blocks, loop bodies, entire loops, or even entire function invocations. If a task contains a function call that expands to many dynamic instructions, the corresponding function definition is considered to be a part of the task. Note that more than one task may contain a call to the same function, in which case the tasks share the static code of the corresponding function definition. Arbitrary control flow and data dependences may exist among instructions of a task or different tasks; specifically, tasks are not necessarily independent. The nonrestrictive nature of tasks allows the Multiscalar architecture to exploit any grain of parallelism, ranging from instructions within a basic block to instructions of different function invocations, present in application programs.

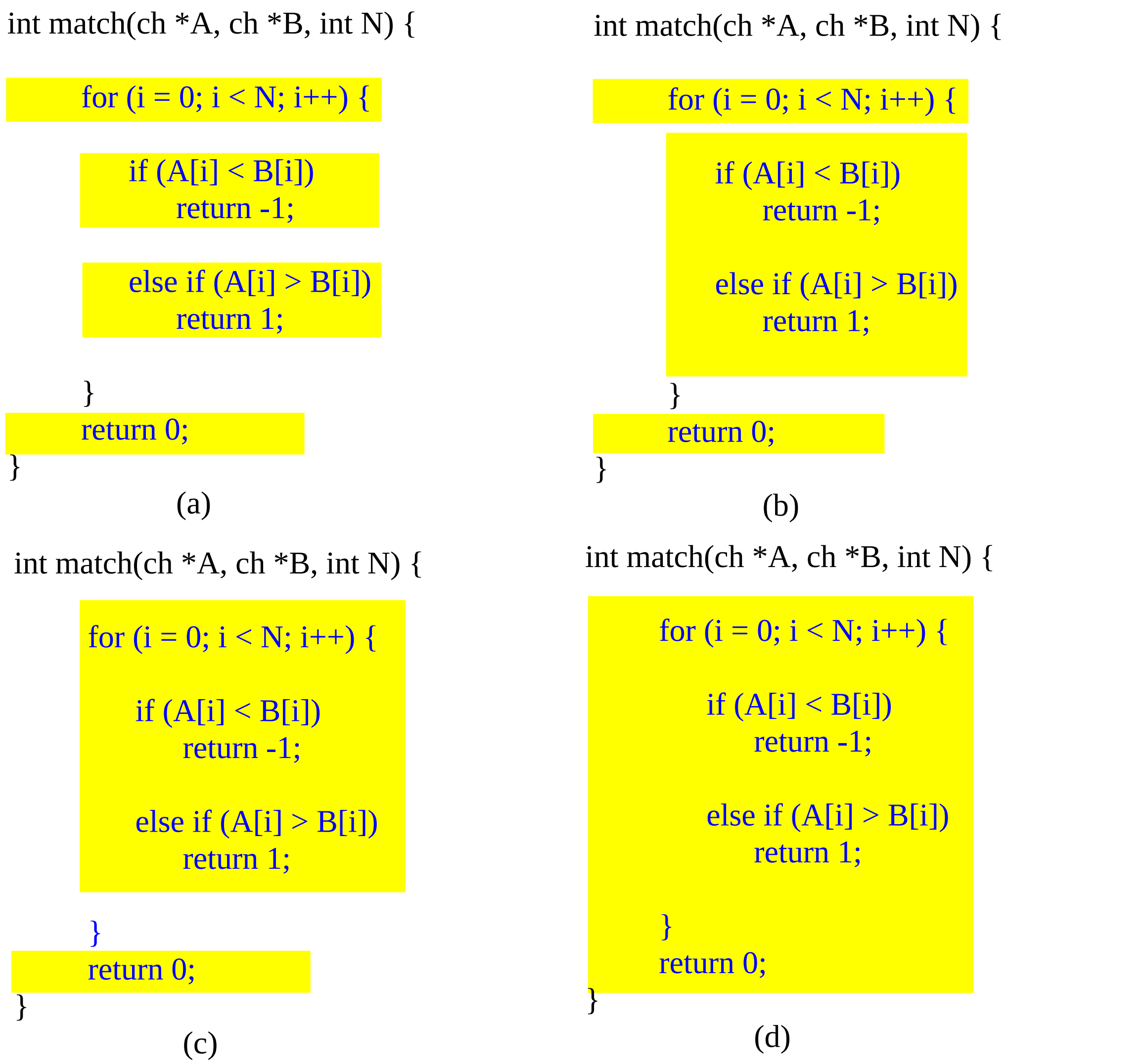
Although tasks are defined to be static objects, there is an important relationship between the sequence of dynamic instructions corresponding to a task executed by a PU and the static task. The PU follows a particular dynamic control flow path through the static task, depending on the data values involved in the computation performed by the task. Since the compiler does not have access to dynamic control flow paths, it treats a set of static control flow paths, some of which give rise to dynamic control flow paths during execution, connected together in a subgraph of the CFG as a task. Thus, a static task, as I have defined it, is inexact in that it contains computation that is a superset of the computa-

[3] Since an instruction is the smallest unit of execution that may be assigned to a PU, each instruction may be defined to be a basic block. But in Chapter 5, which presents experimental results, I use the term basic block tasks to denote the traditional single-entry, single-exit piece of code that may contain more than one instruction.

tion performed by each dynamic invocation of the task. Although inexact, this definition allows the compiler to conveniently perform various analyses and optimizations.

Two simple examples of tasks are: (1) a task that contains the entire program, and (2) a task that contains just one basic block. With such a wide range of options to choose from, task selection uses heuristics to steer towards tasks that deliver high performance. Figure 3-1 illustrates a few examples of Multiscalar tasks indicated by shaded regions, using a simple loop that performs a string compare. Figure 3-1(a) shows the loop body partitioned into three tasks: the first task comprises the case of A[i] < B[i], the second task comprises the case of A[i] > B[i], and the third task comprises increment to *i* and the loop exit branch. Figure 3-1(b) shows the loop body partitioned into two tasks: the first task comprises the comparison of A[i] and B[i], and the second task comprises increment to i and loop exit branch. Figure 3-1(c) shows the entire loop body included in one task and Figure 3-1(d) shows the entire function definition included in one task. These four partitions have different performance characteristics. The partition in Figure 3-1(a) exploits the fine-grain parallelism present within the loop body, the partition in Figure 3-1(b) combines two tasks of Figure 3-1(a) into a larger task, the partition in Figure 3-1(c) demarcates each loop iteration into a task, and the partition in Figure 3-1(d) includes the entire function invocation.

I introduce a few more definitions for the rest of the discussion on Multiscalar tasks. A basic block is defined to be **included** within a task if the basic block is contained in the subgraph of the CFG corresponding to the task. A control flow edge (u,v) of the CFG, where u and v are basic blocks, is defined to be **included** within a task if the basic blocks u and v are both included within the task. A control flow edge (u,v) is defined to be **exposed** if the basic blocks u and v are included in different tasks. Similarly a data dependence edge (p,c) from the producer instruction, p, to the consumer instruction, c, is defined to be **exposed** if any of p, c, and all basic blocks that are in the control flow paths from p to c are included in different tasks.



```
int match(ch *A, ch *B, int N) {

        for (i = 0; i < N; i++) {

            if (A[i] < B[i])
                return -1;

            else if (A[i] > B[i])
                return 1;

        }
        return 0;
}
```

(a)

```
int match(ch *A, ch *B, int N) {

        for (i = 0; i < N; i++) {

            if (A[i] < B[i])
                return -1;

            else if (A[i] > B[i])
                return 1;

        }
        return 0;
}
```

(b)

```
int match(ch *A, ch *B, int N) {

        for (i = 0; i < N; i++) {

            if (A[i] < B[i])
                return -1;

            else if (A[i] > B[i])
                return 1;

        }
        return 0;
}
```

(c)

```
int match(ch *A, ch *B, int N) {

        for (i = 0; i < N; i++) {

            if (A[i] < B[i])
                return -1;

            else if (A[i] > B[i])
                return 1;

        }
        return 0;
}
```

(d)

Figure 3-1: Examples of Multiscalar tasks. Source code of a string matching function. The strings are stored in arrays A and B and the function returns a -1, 1, or 0 depending upon whether A is smaller than, larger than or equal to B, respectively. Each shaded areas corresponds to a task. (a) A task selection containing four tasks: the if portion of the loop body, the else portion of the loop body, the increment of the loop index and the loop-exit test, as shown by the for statement, and the return statement. (b) A task selection containing three tasks: the if and else portion of the loop body, the increment of the loop index and the loop-exit test, and the return statement, as before. (c) A task selection containing two tasks: the entire loop body including the increment of the loop index and the loop-exit test. (d) A task selection containing one task: the entire function definition,

3.2 Performance issues and task characteristics

From the discussion of the task execution time line in Section 2.6, major reasons for performance degradation are: control flow misspeculation, inter-task data dependence, memory dependence misspeculation, load imbalance and task overheads. Each performance issue is related to one or more task characteristic. I now examine each of these categories, distinguish between the categories and their superscalar counterparts, and relate the categories to the task characteristics that cause them.

3.2.1 Control flow misspeculation

Control flow dependences among tasks cause control flow misspeculations. Although control flow misspeculation seems similar to superscalar branch misprediction, the penalty incurred is different. In most superscalar machines, branch prediction is done at every branch one after the other in the predicted path. Typically branches are resolved within a few pipeline stages (usually less than a few tens of cycles) after they are fetched. Consequently, branch misprediction penalties are of the order of a few cycles. But in the Multiscalar architecture, control flow prediction is done well before the actual branch instruction that transfers control from one task to another is even fetched. Resolution of control flow from a task to its successor can be typically done only at the end of the task because the branch instruction may be encountered only at the end of the task. The result of this late resolution is that control flow misspeculation penalties are of the order of the execution time of tasks, which may be much larger than a few cycles of lost opportunity. Even if inter-task control flow may be resolved before the end of the task, misspeculation penalties may still remain a significant fraction of the execution time of tasks.

Figure 3-2 illustrates the impact of control flow speculation on overall performance of the Multiscalar architecture. Figure 3-2(a) shows a part of the CFG of a program. If the program is partitioned so that each of the basic blocks A, B, C, and D is a separate task by itself, then performance may be lost due to many control flow misspeculations.

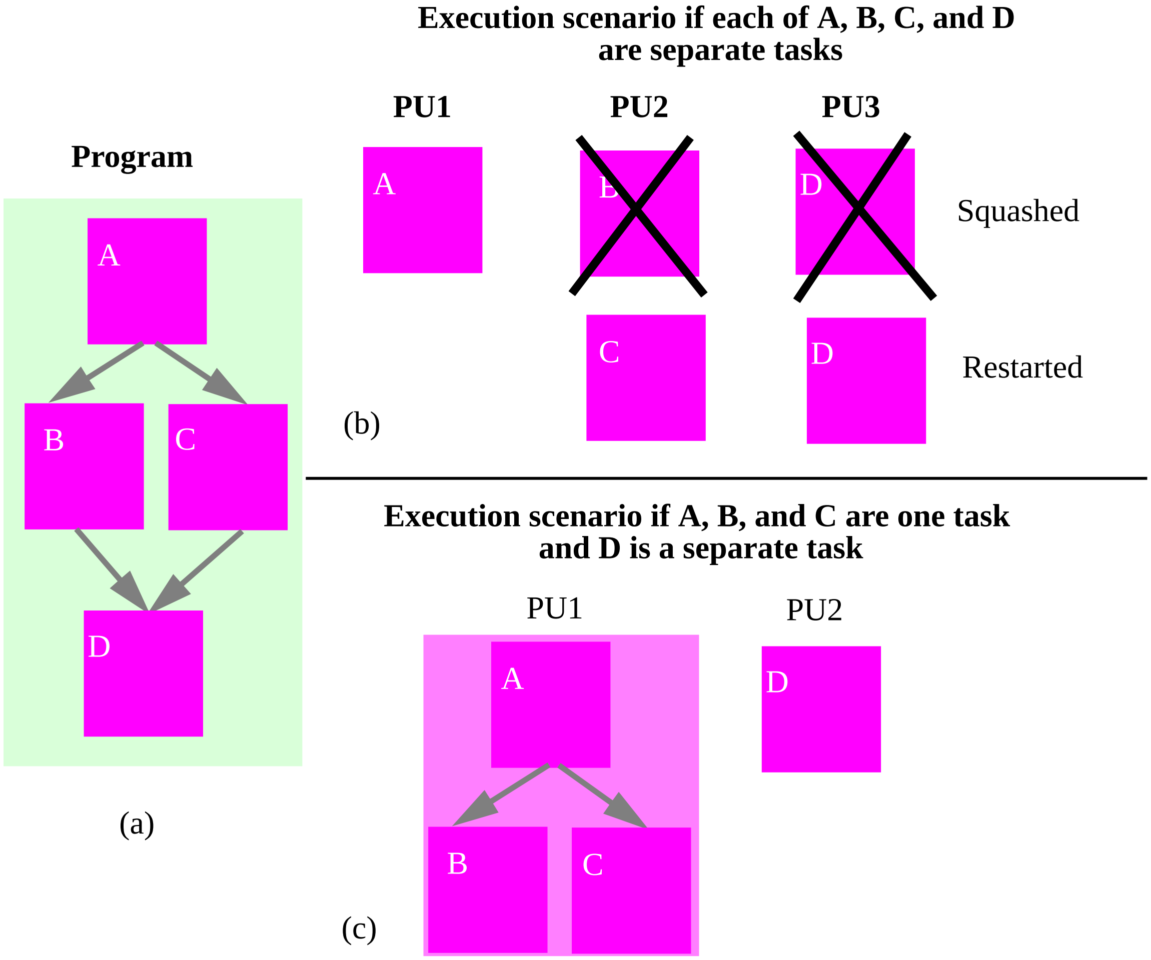

Execution scenario if each of A, B, C, and D are separate tasks
PU1
PU2
PU3
Program
A
B
D
Squashed
C
D
Restarted
A
B
C
D
(b)
Execution scenario if A, B, and C are one task and D is a separate task
PU1
PU2
A
D
B
C
(a)
(c)

Figure 3-2: Impact of control flow. (a) A part of the CFG of a program containing four basic blocks A, B, C, and D. (b) An execution scenario on three processing units, assuming that each of the basic blocks is a task by itself. Control flow speculation assigns tasks A, B, and D to processing units PU1, PU2, and PU3, respectively. Misspeculation is detected and tasks B and D are squashed and tasks C and D are reassigned. (c) An execution scenario on two processing units, assuming basic blocks A, B, and C are included in one task and D is a task by itself. Since the first task has only one successor, there cannot be any control flow misspeculation.

Figure 3-2(b) shows an example of control flow misspeculation during execution. Processing units PU1, PU2, and PU3 are assigned tasks A, B, and D, respectively, via control flow speculation. When control flow is resolved, a misspeculation is detected and the incorrect tasks B and D are squashed and the correct tasks C and D are assigned to execute. The main reason for this misspeculation is that task partitioning exposed the branch at the end of basic block A. If the branch is included within a task, there would be no misspeculation. Figure 3-2(c) shows another task partition which avoids control flow misspeculation. Basic blocks A, B, and C are included within one task and basic block D is in another task. Since Task1 has only one successor since it includes the branch in basic block A, control flow speculation is always correct.

3.2.2 Inter-task data dependence

Inter-task data dependences cause either inter-task data dependence delay or memory dependence misspeculation. Let us consider inter-task data dependence delay first. In superscalar machines, consumer instruction is delayed the most when the corresponding producer instruction is close to it in the dynamic instruction stream. The delay experienced by the consumer is typically proportional to the latency of the producer. On the other hand, in the Multiscalar architecture, the consumer may experience delays much longer than the latency of the producer if the producer and the consumer belong to different tasks. If the producer is executed at the end of its task and the consumer is executed at the beginning of its task, then the delay incurred is proportional to the execution time of the producer task. Data dependence that may lead to only a few cycles delay in a superscalar machine may get aggravated to much longer delay if such a dependence is spread out across large tasks. Figure 3-3 illustrates the impact of data dependences on overall performance of the Multiscalar architecture. Figure 3-3(a) shows a loop in which the variable Y is defined and then used. In Figure 3-3(a), the loop body is partitioned into two tasks, A and B, resulting in the data dependence edge from the define of variable Y to the use of variable Y to be exposed. The tasks are assigned to execute on processing units PU1 and PU2, as shown in Figure 3-3(b). During execution, PU2 waits for the value of Y which is produced at the end of task A, reducing the overlap of execution on PU1 and PU2. The idling of PU2 results in loss of performance. Figure 3-3(c) shows an execution scenario similar to

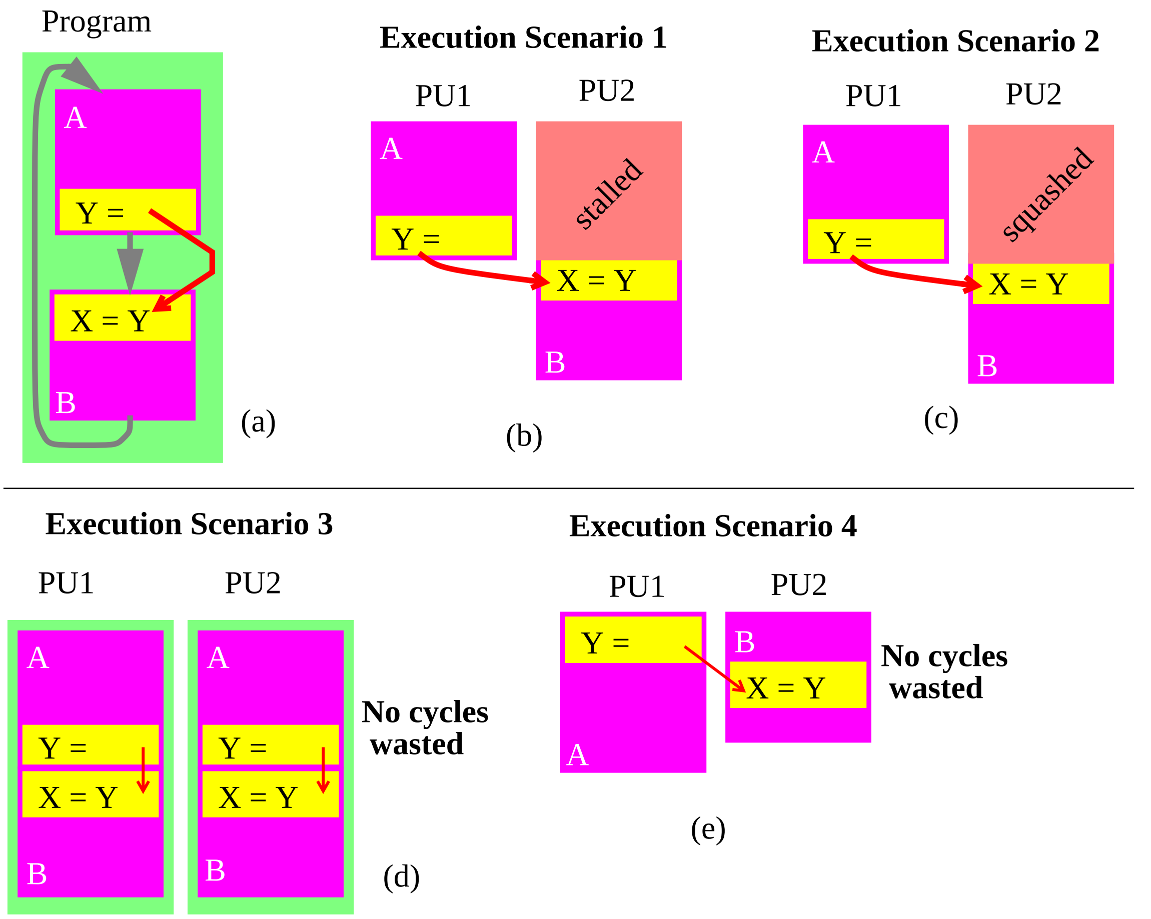


Figure 3-3: Impact of data dependences. (a) A part of the CFG of a program containing a loop with a definition and a use of the variable Y. The loop body contains two parts shown by shaded regions A and B. ➡ indicates control flow. → indicates data dependence. (b) An execution scenario on two processing units, assuming that regions A and B are separate tasks. Tasks A and B are assigned to processing units PU1 and PU2, respectively. Since task B uses the value of Y at the beginning, it is stalled until task A produces the value. (c) An execution scenario similar to that in (b), except instead of waiting for the value, task B proceeds under data dependence speculation that there is no data dependence. When task A produces the value, a data dependence misspeculation is detected and task B is squashed and restarted. (d) An execution scenario on two processing units, assuming that the entire loop body, i.e., regions A and B are included in one task. Two iterations of the loop are assigned for execution on the two processing units. Since the data dependence is included within the task, the two tasks proceed without any stalling or squashing. (e) An execution scenario similar to that in (b), except the value is produced early in task A and consumed late in task B. Tasks A and B proceed without any stalling or squashing and task A produces and sends the value before task B needs to use it.

Figure 3-3(b) but instead of waiting for the value of Y, PU2 speculates that there is no data dependence between tasks A and B and uses a stale value of Y; when PU1 produces a new value for Y, a data dependence misspeculation is detected and PU2 is squashed and restarted. Irrespective of whether PU2 waits or misspeculates, the inter-task data dependence results in a loss of performance. Figure 3-3(d) shows the entire loop body included in one task so that the data dependence edge is included within the task. Since no data dependence edge is exposed, the two processing units execute simultaneously, achieving better performance. Figure 3-3(e) illustrates the relationship between task partitioning and scheduling of inter-task data dependence. If the define and the use of Y are scheduled so that the define appears early in the schedule of the instructions of task A and the use appears late in the schedule of the instructions of task B, then even if the data dependence is exposed, it will not cause any loss of performance. Figure 3-3(e) shows tasks A and B assigned to processing units PU1 and PU2, respectively. Unlike the scenario in Figure 3-3(a), PU1 produces the value of Y well before PU2 uses the value and the communication due to the inter-task dependence is completely overlapped with useful computation.

If an inter-task data dependence is misspeculated (instead of waiting), then, again similar performance loss is incurred. Misspeculation is detected when the producer instruction executes and the required value is sent to the consumer instruction; the consumer signals the occurrence of a data dependence violation and a squash is triggered. If the producer is executed at the end of its task and the consumer is executed at the beginning of its task, then the squash penalty incurred is proportional to the execution time of the producer task. Here, again, a few cycle delay in a superscalar machine may get aggravated to much larger squash penalties.

3.2.3 Load imbalance

When a PU completes executing a task, the PU may commit the register and memory state modified by the task only after the predecessor task has been completed and committed. Committing tasks in program order is a straightforward way to ensure sequential semantics. If a small task follows a large task in program order, then the PU executing the small task idles after completing the small task, waiting for the large task to complete. This loss of performance arises primarily because of load imbalance, i.e., the amount of computation performed in the two tasks are not equal. For example, let us assume that two tasks executing on adjacent processing units take 20 and 10 cycles, respectively. On the average, 50% of the time the second processing unit is idle, resulting in overall utilization of 75%.

Load imbalance causes performance loss for large scale parallel machines [68]. Since distributed processor organizations also partition their physical resources among tasks similar to parallel machines, they are confronted with load imbalance problems. Large variations in the amount of computation of adjacent tasks causes load imbalance resulting in successor tasks waiting for predecessor task to retire. A task assigned for execution may only use its own PU; resources of other PUs can not be used even if they are free. To the first order, load imbalance results from large variations in the number of dynamic instructions executed by tasks assigned to adjacent processing units. Since superscalar machines employ centralized shared resources, an instruction needing a resource can utilize it if it is free. Load imbalance can be alleviated by allowing new tasks to execute on PUs that have completed their old tasks; the speculative state of the old tasks have to be buffered separately from the new tasks and committed appropriately, which may complicate the buffering scheme. Similar to the previous categories, cycles wasted due to load imbalance also may be proportional to the execution time of tasks.

3.2.4 Task overheads

There are two kinds of overheads associated with tasks: (1) task start overhead, and (2) task end overhead. Task start overhead is similar to the problem of short vectors in a vector machine [87]. Small tasks incur task start overheads to be a significant fraction of their execution time resulting in considerable performance loss. Unlike the other categories, task start overhead depends on hardware parameters of PUs (instruction fetch unit latencies, number of pipeline stages for fetch and decode, etc), although it is significant only if tasks are small. Overlapping the start of another task with the execution of a task may reduce the overhead but may complicate the design of the PUs.

At the end of a task, all speculative state of the task is committed to the architectural storage; if task commits involve actual movement of data (as opposed to just tagging of data as committed without physical movement) extra cycles are spent. Depending upon the amount of data that is moved, a significant percentage of the cycles required to execute the task may be spent in committing its state. Tagging of data to avoid physical movement may complicate speculative state management. Large tasks typically have large amount of speculative state to be committed.

If tasks contain a large number of dynamic instructions, then several problems arise: (1) Large tasks increase both the number of misspeculations and the penalty associated with each misspeculation. Large tasks typically contain a large number of memory instructions and speculating over a large number of memory operations usually results in an increase in misspeculations because there is a large likelihood of misspeculating a true data dependence. When a large task gets squashed, a large amount of work gets thrown away resulting in large squash penalties. (2) Large tasks cause the buffers that hold the speculative state to fill up causing the task to stall until the speculation is resolved. In the case of the ARB, this filling up is called ARB overflow. (3) Large tasks result in a loss of opportunity to exploit the parallelism within them because tasks execute on narrow PUs that may not be wide enough to put large amounts of intra-task parallelism to use.

3.2.5 Summary: Task characteristics that affect performance

Based on the above discussion, it can be seen that task characteristics that affect performance are: inter-task control flow, inter-task data dependences and task size.

Control flow misspeculation is affected by the predictability and the position (dynamically encountered/executed early or late during the execution of the task) of the control flow dependences that are exposed by task selection. Inter-task data dependence delay is determined by the position (dynamically encountered/executed early or late during the execution of the task) of the data dependences that are exposed by task selection. Load imbalance is primarily due to variance in dynamic task sizes. Task overheads are influenced by task size; task start overhead is due to short tasks and task end overhead is due to long tasks with a large amount of speculative state.

These parameters are used to navigate task selection. Ideally, task selection should be guided by a precise timing model of the underlying Multiscalar hardware so that the tasks that achieve the highest performance (as predicted by the model) may be selected. But there are several problems that cause a deviation from this ideal: (1) An accurate timing model of the hardware is hard to construct if the hardware includes many dynamic components like out-of-order issue, many levels of cache hierarchy, branch prediction and arbitrated queues and buses. (2) Even if an accurate model is available, selecting the tasks that achieve the highest performance as predicted by the timing model is an intractable problem.

Instead of using a timing model to guide task selection, I use a heuristic for each characteristic. Each heuristic attempts to improve the task characteristic it is designed for and all the heuristics are combined together in one task selection process. Any conflicts among the heuristics are resolved by prioritizing the heuristics. The reasons for the choice of the heuristics implemented and the chain of arguments that channel the heuristics in a progression are given later in Section 3.4.

3.3 Task selection criteria

Before describing the actual heuristics that I have implemented, I discuss some general criteria for options available to the compiler to control task characteristics and select suitable tasks. I group task selection criteria based on the task characteristic they influence. I discuss criteria corresponding to task size, control flow, and data dependence.

3.3.1 Task size criteria

Tasks can be neither large nor small. As discussed before, small tasks may incur high start-up overheads and large tasks may cause memory dependence squashes and ARB overflow. I discuss some of the options available starting from simple tasks each of which comprise a single basic block.

Basic blocks are well-known structures used by compilers in various analyses. Unfortunately, basic block tasks may not achieve high performance for several reasons. For many application programs, basic blocks may not contain enough instructions to offset task overheads. Also, most register dependences usually extend beyond a single basic block causing basic block tasks to wait for register values. Although basic block tasks usually have only two successors, which may not tax the prediction hardware, they may expose unpredictable branches to the prediction mechanism.

Multiple basic blocks that comprise a subgraph of the CFG can be assembled into tasks. Since there no correctness constraints on how many basic blocks may be included in a task, tasks comprising arbitrarily many basic blocks can be generated. But, generating tasks comprising many basic blocks may lead to the problem of large tasks.

Apart from including many basic blocks within tasks, including entire loops and function invocations within tasks also may lead to large tasks. Terminating tasks at function invocations as well as entry and exit of loops naturally limits task size. Most modular applications tend to have function calls, which naturally prevent large tasks. But frequent function invocations with little computation between them and loops with small loop bodies may lead to small tasks. Loops can be unrolled so that multiple iterations of short loops can be included to increase the size of short loop-body tasks; short function invocations can be included entirely within tasks to avoid small tasks. Including or not including a function invocation permits a limited degree of freedom (all-or-nothing) to handle function invocations in task selection policy. A better range of options are possible by inlining function invocations and selecting tasks out of the merged computation.

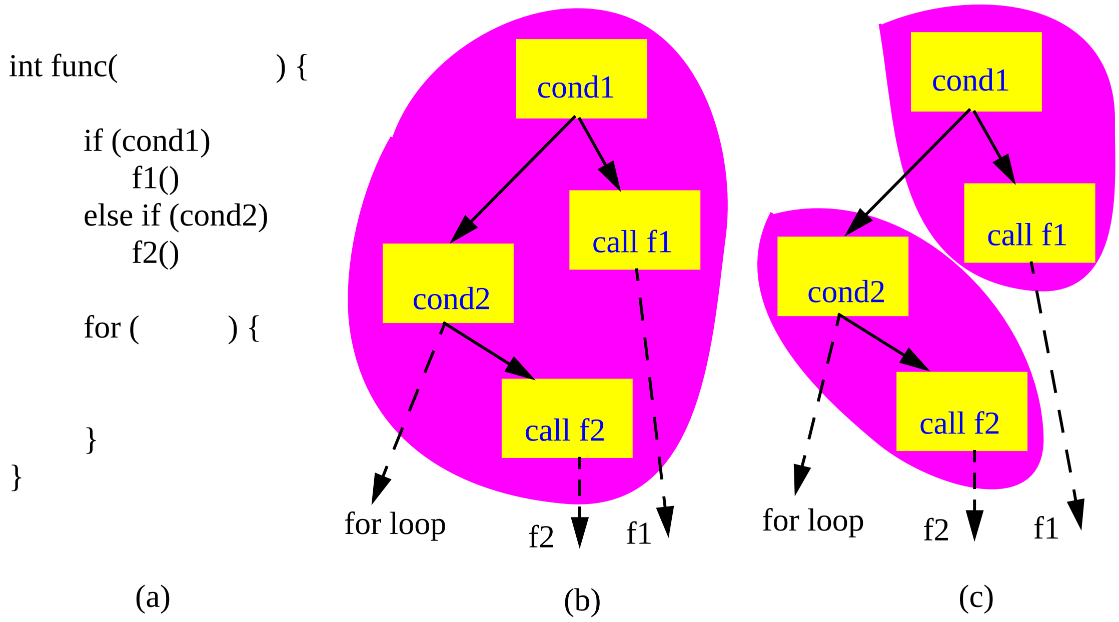
The criteria used to control task size may generate tasks with varying sizes (number of dynamic instructions) leading to load imbalance problems. To alleviate load imbalance problems, variation in task size may be controlled by bounding the maximum task size to a predefined threshold. Task selection process can control task size[4] by generating tasks whose size does not cross the threshold. In the presence of complex control flow within tasks, estimating task size is more involved than counting the number of static instructions. Dynamic instruction count, based on profiling information, may be used as an estimate for task size.

3.3.2 Control flow criteria

Since a task is assigned for execution via control flow speculation, it is important that tasks are selected with as many successors as can be tracked by the hardware prediction

[4] if control flow path frequency information is available via profiling then the dynamic size may be monitored; otherwise, the static size is monitored.

tables. These hardware tables are built to track some fixed number (N) of successors. If tasks are selected to have more than N successors, dynamic control flow speculation accuracy may decrease, leading to loss of performance. In most structured applications, although the CFG of a subroutine diverges at the beginning due to conditional statements, control flow paths reconverge eventually before the end of the subroutine. Therefore, constraining the number of successors may seem trivial. But imposing task size criteria may result in a task being terminated at loops and function invocations, which are **default successors** of the task; the top of the loop corresponding to a loop back edge and the invoked function corresponding to a function invocation are examples of such default successors. Figure 3-4 illustrates default successors. The task shown in the figure has three default



```
int func(                ) {

        if (cond1)
             f1()
        else if (cond2)
             f2()

        for (          ) {


        }
}
```

Figure 3-4: Number of successors. (a) Parts of the source code of a function. (b) Part of the CFG corresponding to the if-else-if statement. The dashed arrows represent exposed control flow edges and shaded regions represent tasks. The shaded region shows a task that includes all the four basic blocks and has three successors: f1, f2, and the for loop. (c) If only two successors are allowed, then a possible partition with two tasks is shown.

successors: function f1, function f2, and entry into the for loop. If the number of targets tracked by the hardware is less than three, then the task may have to be split into two smaller tasks with two successors each, as shown in Figure 3-4(c).

The number of default successors of a task may be more than N, which may lead to control flow misspeculations. But generating tasks with a few successors may lead to small tasks. For example, if tasks are restricted to have only two successors, then tasks mostly contain just a single basic block. So, controlling the number of successors even while including multiple basic blocks within tasks is an important constraint for task selection process. Task selection process can track the number of successors of tasks generated so that tasks are formed with at most N successors. If a task has more than N default successors, then the task can be split into smaller tasks such that the smaller tasks have at most N successors. While controlling the number of successors, task selection process can take advantage of reconvergent control flow paths and branches that are easier to predict than others.

Reconvergent control flow paths (e.g., the if-path and the else-path of a conditional statement) are advantageous in two ways: (1) A branch can be included within a task (i.e., the task contains both the taken path and the not-taken path of the branch until they reconverge). The included branch does not affect inter-task control flow and cannot cause control flow misspeculations, and (2) By utilizing reconvergent paths, the size of a task can be increased without taxing the prediction hardware with larger number of successors.

One of the advantages of task level prediction over ordinary branch prediction is that not every branch in the program is predicted; only those branches that form task boundaries are predicted and branches internal to a task do not affect the prediction process. Branches that are internal to a task do not affect the inter-task prediction process. Branches that are easier to predict (e.g., the exit-test branch of a loop) can be exposed (defined in Section 3.1) and branches that are harder to predict (e.g., data dependent branches) can be included within tasks. The task including the entire loop body in Figure 3-1(c) is an example of a task which includes data dependent branches.

3.3.3 Data dependence criteria

Data dependences from producer instruction contained in one task to consumer instruction contained in another task manifest as inter-task data dependences from producer task to consumer task. If a dependence is included within a task, then no inter-task communication is required to honor that dependence. Intra-task data dependences, i.e., data dependences among instructions within a task, do not impact performance more than the usual pipeline stalls. Inter-task data dependences that extend beyond the window established by the hardware (i.e., the producer task executes and commits before the consumer task is predicted and assigned for execution), also do not impact performance.

The main cause of performance loss due to communication delay is that instructions that are involved in data dependences, namely the producer instruction and the consumer instructions, are not scheduled favorably; producer instructions need to be executed early and consumer instructions need to be executed late in their respective tasks. With respect to the time a task starts to execute, the time a producer instruction in the task is executed depends on (1) whether the instruction depends on values from predecessor tasks and if so, the time the values get produced and are communicated and (2) how many other instructions in the task are ahead of the instruction as per program order, since instructions in a task are executed on a PU in the usual uniprocessor style of execution. A producer instruction, therefore, may execute much later than the start of the task if the values it needs are available later or if the task contains many other instructions that precede the instruction.

There are many ways to handle data dependences during task selection: (1) All data dependences are included within tasks and tasks selected are completely data independent, (2) the program is partitioned so that all inter-task dependences extend beyond the hardware window, and (3) the program is partitioned so that all inter-task dependences are scheduled perfectly, i.e., the producer of the inter-task dependence executes before the consumer does and the value reaches the consumer before it executes, so that the consumer does not stall.

There are several impediments to achieving the ideal selection: (1) Several memory dependences are unknown or ambiguous at compile-time, (2) including a data dependence within a task (as defined in Section 3.1) may result in a task with more successors than desired, and (3) partitioning the program so that all inter-task dependences extend beyond the dynamic window may result in tasks that are too small[5]. Figure 3-5 illustrates task partitions that may result when a data dependence edge is considered. Figure 3-5(a) shows a part of the CFG of a program including a data dependence edge from the top basic block to the bottom basic block. For this example, let us assume that the number of hardware targets is 4. Figure 3-5(b) shows a task that includes the data dependence edge by including all the basic blocks in the control flow path from the producer basic block to the consumer basic block. But the task has five successors which exceed the number of hardware targets (assuming each of the control flow edge that is exposed leads to a different successor). Figure 3-5(c) shows a task partition in which each basic block is a task of its own. If a hardware configuration contains four processing units, then the data dependence edge extends beyond its dynamic window of instructions. But the tasks are small.

Exposing a data dependence to avoid exceeding the number of successors may be inevitable in certain cases. One way of alleviating performance problems due to an exposed data dependence is to avoid including many preceding instructions in the same task as the producer instruction, involved in the dependence. Similar arguments may be made for consumer instructions as well. Task selection, thus, may enable favorable scheduling of inter-task communication by controlling the choice of basic blocks that are included in a

[5] The compiler may have to be aware of the number of PUs used in the hardware configuration, which makes transparent performance scaling of the hardware difficult.

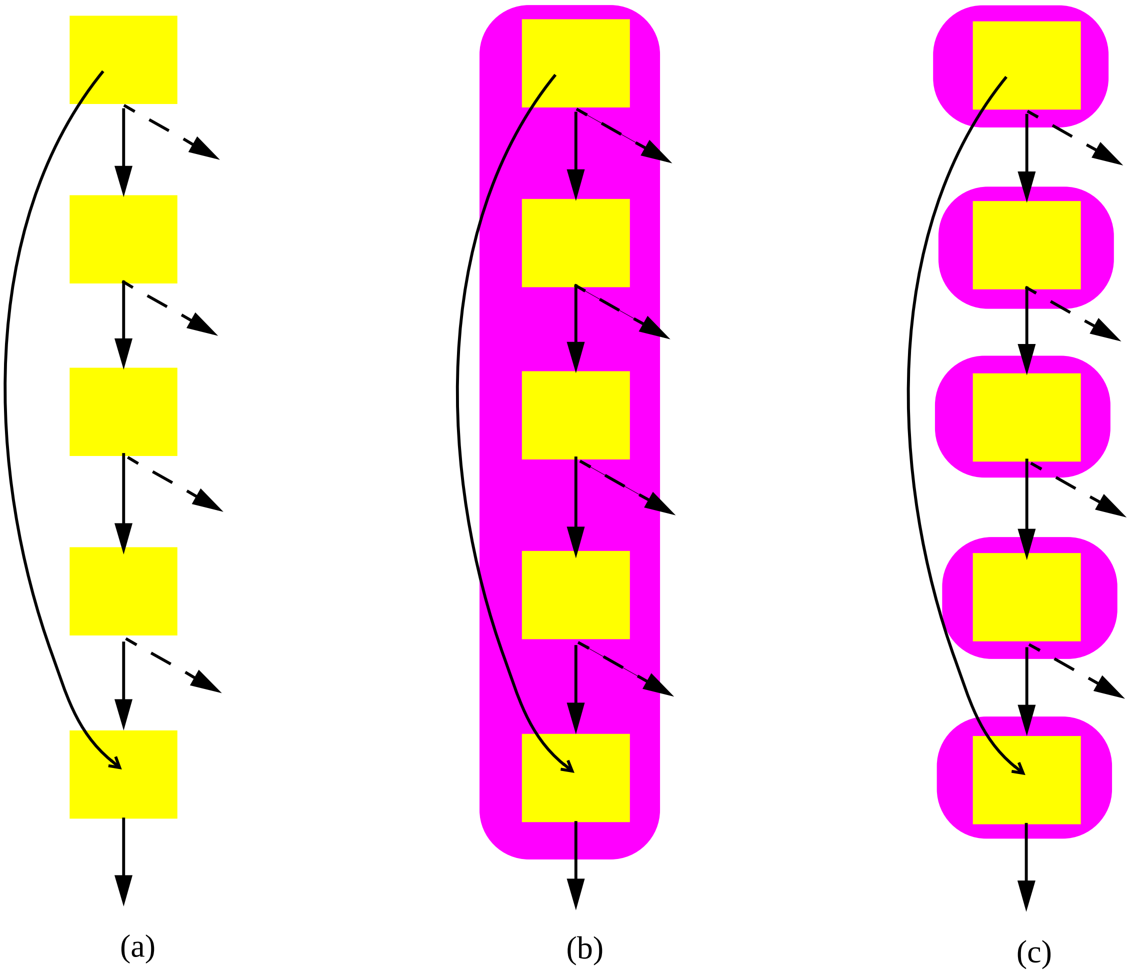


Figure 3-5: Task selection and data dependence edges. (a) A part of the CFG of a program. ➤ indicates a control flow edge and ⟶ indicates a data dependence edge. Dashed arrows represent terminal control flow edges and shaded regions represent tasks. (b) A task selection resulting in one task that includes the data dependence edge and has five successors. (c) A task selection resulting in five tasks each of which has two successors. The data dependence edge is not included within a task but if the hardware contains only four processing units then the data dependence edge extends beyond the hardware window of instructions.

task. Figure 3-6 illustrates how task partitioning may affect inter-task data dependences. Figure 3-6(a) shows a part of a CFG with a data dependence edge from the top basic block to the bottom basic block. Figure 3-6(b) shows a task partition obtained by control flow

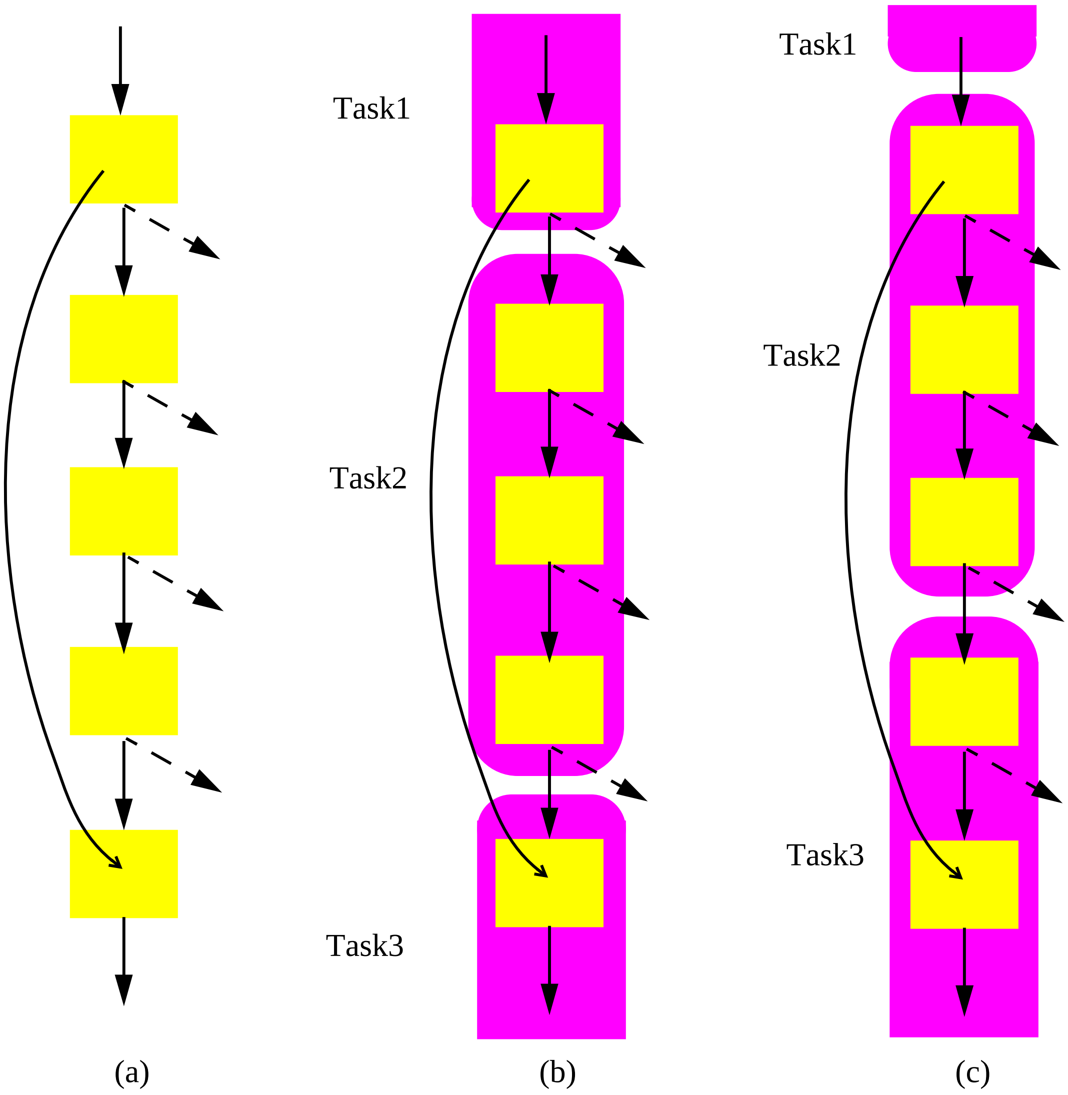


Figure 3-6: Task selection and inter-task data communication scheduling. (a) A part of the CFG of a program. ➤ indicates a control flow edge and ⟶ indicates a data dependence edge. Dashed arrows represent terminal control flow edges and shaded regions represent tasks. (b) A task selection resulting in three tasks. The producer of the data dependence is at the end of Task1 and the consumer is at the top of Task3, which leads to Task3 waiting for the value from Task1 during execution. (c) A task selection resulting in three tasks. The producer of the data dependence is at the top of Task2 and the consumer is at the bottom of Task3, causing the value to be communicated from Task2 to Task3 before Task3 needs it.

heuristics, assuming the number of hardware successors to be four. Since control flow heuristics do not take data dependences into consideration, the producer of the data dependence is included at the end of Task1 and the consumer is included at the beginning of Task3, aggravating data dependence delay. Figure 3-6(c) shows a task partition obtained by data dependence heuristics consisting of two tasks in which although the data dependence edge is exposed, the tasks are such that the resultant inter-task communication is scheduled favorably; the producer instruction is placed early in its task at the first basic block of the task and the consumer instruction is placed late in its task at the last basic block of the task. If the data dependence edge is included within one task then the task gets five successors exceeding the number of hardware targets of four, as shown in Figure 3-5(c). Data dependence heuristics, instead, partition the CFG such that the producer of the data dependence is at the top of Task2, facilitating communication of the dependence value without significant delay. Moreover, the tasks have three successors each, which are fewer than the number of hardware targets.

Task selection can track register and simple memory dependences and include such dependences within tasks. If a basic block starts a data dependence chain, then the entire dependence chain can be included within a task; if, for reasons mentioned above, the entire dependence chain cannot be included or a data dependence recurrence (e.g., a linear recurrence through a loop) is detected, then the dependence chain can be broken into several tasks. Each of the data dependence edge from the chain can be exposed such that the producer instruction is positioned early in its task.

3.3.4 Interaction among the criteria

Each of the criteria discussed above may favor different task selections. It is difficult to estimate the performance benefits of each of them separately and then select the best. In practice, key conflicts occur between control flow loop criteria and data dependence criteria. Data dependence criteria may partition a loop body into several tasks because the loop body contains several independent chains of computation. But control flow criteria may demarcate the entire loop body as one task, causing iterations of the loop to overlap. Although control flow criteria do not take advantage of the parallelism present within the loop body, the single loop body task may perform better than multiple loop body tasks because its control flow speculation accuracy is significantly higher and load imbalance is considerably less. Data dependence tasks incur control misspeculations because they expose branches that are hard to predict, as opposed to the loop back branch exposed by control flow criteria. In addition to control misspeculations, data dependence tasks may not be load-balanced due to natural variations in the amount of computation involved in data dependence chains; loop iterations, on the other hand, may be naturally better load-balanced.

3.4 Task selection heuristics

In this section, I describe heuristics to select tasks based on the criteria described above. I start with the basic process of CFG traversal used by task selection. Then I discuss how the heuristics are incorporated to steer the traversal. The heuristics were implemented in a progression starting with tasks containing single basic block. In an attempt to alleviate the performance problems caused by the small size of basic block tasks, multiple basic blocks were included within tasks. But tasks containing multiple basic block tasks incurred excessive inter-task control flow misspeculations. To mitigate control flow misspeculations, the number of successors of a task were controlled through control flow heuristic. Even though control flow speculation was improved, inter-task data dependences were aggravated, resulting in performance loss. To curb this loss, data dependences were included within tasks through data dependence heuristic. The heuristics are integrated together so that suitable tasks are carved out of programs.

3.4.1 Basic task selection process

Task selection proceeds by traversing the CFG of the application starting at the root of the CFG. Basic blocks are included in a task by progressively examining whether successors of the basic blocks that have already been included in the task may also be added to the task. The heuristics are incorporated in the selection process via decision-making functions that determine whether a particular basic block should be included in a task or not. If the heuristics terminate a control flow path by not including a basic block, then another task is grown starting at the basic block. The basic block at which each task is entered is called the root of the task.

Demarcating basic blocks as tasks is easy for the compiler because basic blocks are already identified. Tasks so obtained are called **basic block tasks**. No special decision-making functions are required to generate basic block tasks. Each basic block is the root of its task.

3.4.2 Control flow heuristic

To overcome some of the performance problems of basic block tasks, programs are partitioned into tasks containing multiple basic blocks. Tasks so obtained are called **control flow tasks**. If multiple basic blocks are included within a task, then a decision-making function that decides whether to include a particular basic block within a task or not is needed. To this end, basic blocks and control flow edges are categorized as **terminal** or **non-terminal**. If a basic block is terminal, then none of its successors (in the CFG) are included in the task that contains the basic block. Terminal edges are those which are not included within a task (i.e., if (u,v) is a control flow edge, then u and v are not included in the same task). Non-terminal edges may or may not be included within a task, depending upon the heuristics. Loop back edges and edges that lead into a loop, and basic blocks that end in a function call or a function return are terminal. For example, Figure 3-4(b) shows a subgraph with three terminal control flow edges leading to three successors.

During the CFG traversal, if any terminal edges or terminal basic blocks are encountered, then the path is not explored any further and the basic blocks that terminate the path are marked as successors. In order to ensure that tasks have at most N successors, the number of successors is tracked when basic blocks are included within tasks; the largest collection of basic blocks that correspond to at most N successors called **the feasible task** is also tracked. After a basic block is added to the task, if the resulting number of successors is at most N, then the basic block is added to the feasible task. By taking advantage of reconvergent control flow paths, tasks are made larger without necessarily increasing the number of successors. But during the traversal, it is not known a priori which paths reconverge and which do not. Control flow heuristic uses a greedy approach; the traversal continues to explore control flow paths even if the number of successors exceeds the allowed limit. When all the control flow paths are terminated, the feasible task so obtained demarcates the task.

To determine the importance of controlling the number of successors, I experimented with tasks containing multiple basic blocks without bounding the number of successors. Control flow speculation accuracy of such tasks is significantly low, affecting performance adversely because most of the tasks have a large number (> 10) successors, which taxes the prediction mechanism. Overall performance degrades considerably, making this approach inferior, compared to control flow heuristic.

3.4.3 Data dependence heuristic

The key problem with exposing data dependences is that if the producer is encountered late and the consumer is encountered early, then many cycles may be wasted waiting for the value to be communicated. The main goal of data dependence driven task selection is that for a given data dependence extending across several basic blocks, either the dependence is included within a task or it is exposed so that the exposed dependence does not cause communication delays.

In order to alleviate performance loss due to inter-task register dependences, data dependences are identified and programs are partitioned such that data dependences (register dependences and memory dependences that can be disambiguated using simple schemes) that span multiple basic blocks are included within tasks. Tasks so obtained are called **data dependence tasks**. Since control flow task heuristics already contain mechanisms to select tasks that contain multiple basic blocks, data dependence task heuristics build on control flow task heuristics. During the selection of a task, data dependence heuristics steer the exploration of control flow paths to those basic blocks that are dependent on the basic blocks that have been included in the task. Control flow heuristics include basic blocks in tasks regardless of whether they are dependent on other basic blocks contained in the task. Data dependence heuristics, instead, include a basic block only if it is dependent on other basic blocks included in the task. Thus, data dependence heuristics explore only those control flow paths that lead to dependent basic blocks and terminate the other paths.

Data dependences are determined using traditional static analyses. Register dependences and simple memory dependences (involving named scalar variables) are established by standard dataflow analyses [1] [3] [12] [13] [19] [20] [24] [25] [50] [57] [59] [60] [69] [74] [75] [84] [85] [88]. Data dependence information restricted to identifying the producer and the consumer instruction is not sufficient to include the dependence within a task. If the producer and the consumer instructions are not in adjacent basic blocks in the control flow graph, then other basic blocks in the control flow path from the producer to the consumer also have to be included. Therefore, the compiler identifies the set of basic blocks that have to be included in a task so that a data dependence may be included within the task. This set of basic blocks is called the **codependence set**. Figure 3-7 illustrates inclusion of a data dependence within a task. Figure 3-7(a) shows a task which includes the data dependence edge with five successors. The task contains only the basic blocks that are in the codependence set of the data dependence. By including a few more basic

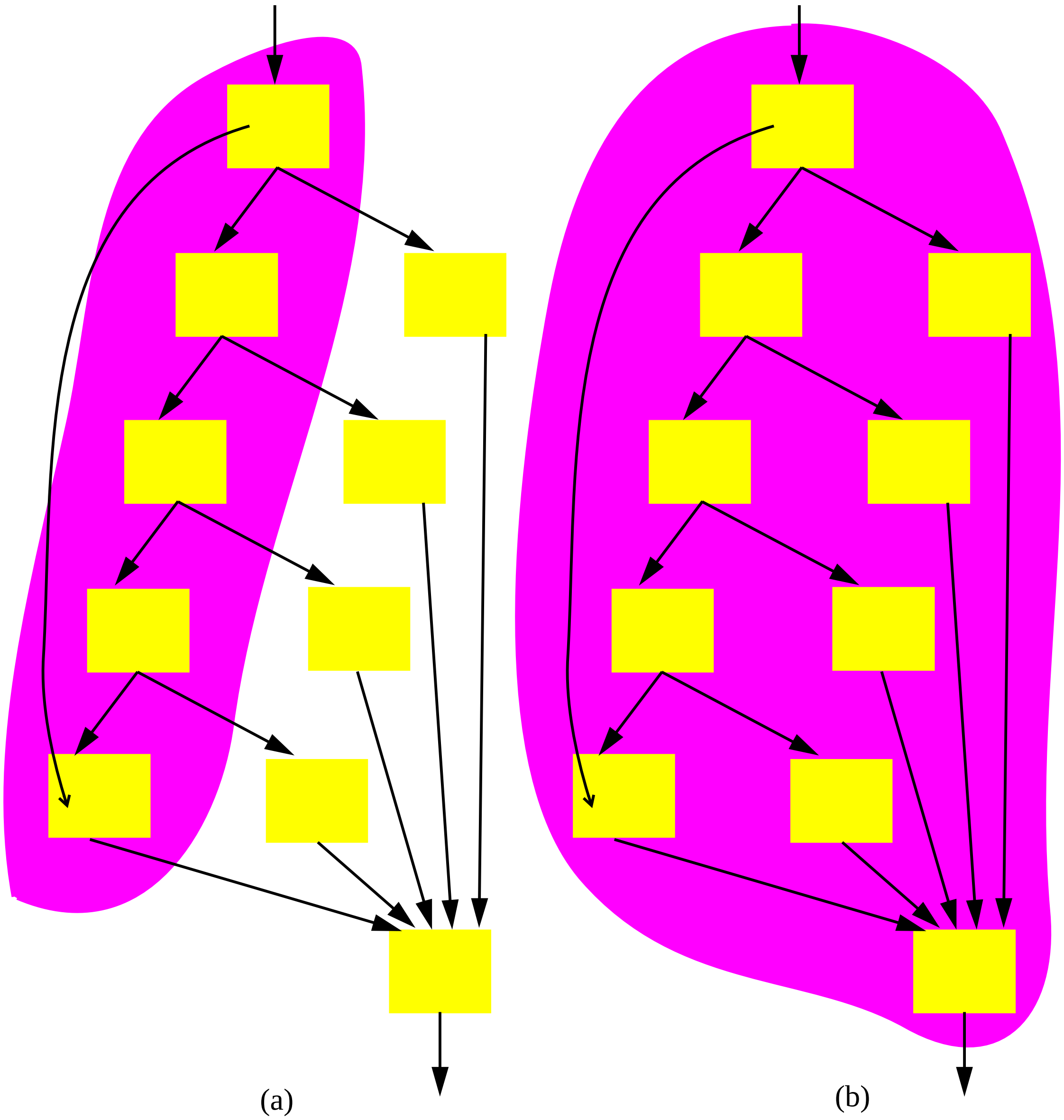


Figure 3-7: Including a data dependence within a task. (a) A part of the CFG of a program. ➤ indicates a control flow edge and ⟶ indicates a data dependence edge. Shaded regions represent tasks. The task includes the data dependence edge but has five successors. (b) By including more basic blocks in the task, the number of successors is reduced to one. The task includes the data dependence and has only one successor.

blocks which may not be in the codependence set within the task, the number of successors may be reduced, as shown in Figure 3-7(b).

The set of dataflow equations, given in Figure 3-8, identify the set of basic blocks that have to be included: the basic blocks that lie in the control flow paths from the producer to the consumer. The equations for *DefReach* compute the basic blocks in which the definition of the register *r* by the producer is reached. *DefReach* is true at a basic block for a register if it is true at one of its parents and the parent does not overwrite the register. *DefReach* identifies all those basic blocks in which the definition from the producer is reached, not necessarily only those that lie in control flow paths between the producer and the consumer. To get the restricted set of basic blocks for which *DefReach* is true and which lie in control flow paths between the producer and the consumer, the dataflow property *Codependent* is used. The dataflow property *Codependent* defined for a register *r* and

$$\mathrm{DefReach}(r, \mathrm{Producer}) = \mathrm{TRUE}$$

$$\mathrm{DefReach}(r, c) = \sum_{p \in \mathrm{parents}(c)} \mathrm{DefReach}(r, p) \cap \overline{\mathrm{BBdef}(r, p)}, \ c \neq \mathrm{Producer}$$

$$\mathrm{Codependent}(r, \mathrm{Consumer}) = \mathrm{TRUE}$$

$$\mathrm{Codependent}(r, p) = \sum_{c \in \mathrm{children}(p)} \mathrm{Codependent}(r, c) \cap \mathrm{DefReach}(r, p)$$

Figure 3-8: Data dependence heuristics. BBdef(i,r) is true if register r is modified in basic block i. *DefReach(r, b)* is true for any successor b of basic block Producer if the definition of register r reaches b without being overwritten by any other definition of r. *Codependent(r, p)* is true for any predecessor p of basic block Consumer if *DefReach(r,p)* is true.

a basic block is true for those basic blocks for which *DefReach* is true and which lie in control flow paths between the producer and the consumer. If all of the set of basic blocks is included within the task, then the dependence is included within the task completely. If any of the set of basic blocks are not included in the task then some control flow path from the producer to the consumer is exposed, resulting in the dependence to be exposed. If the set of basic blocks contribute more than N successors, then the task is further expanded by exploring those control flow paths that were terminated because they did not lie between the producer and the consumer. If the expansion decreases the number of successors to N then the dependence is successfully included; otherwise the dependence cannot be included, and to facilitate favorable scheduling of the resultant inter-task communication, the basic block containing the producer is marked to be the root of its task.

There is a trade-off between control speculation accuracy and data dependence delay illustrated by those dependences that if included result in more than N successors: If the task is made to include the dependence, then there is no inter-task communication delay (since there is no inter-task communication) but control speculation accuracy may be worsened due to the larger number of successors; otherwise, control speculation accuracy is not affected but inter-task data communication delay may be worsened.

The process described above produces tasks that include or not include one data dependence. If more than one dependence is taken into consideration, then including one dependence may exclude another because inclusion of a certain dependence may result in some control flow paths to be terminated and inclusion of another dependence may require some of the previously terminated paths to be not terminated. A simple solution to this difficulty is to prioritize the dependences and favor including dependences of higher priority before considering lower priority dependences. A simple prioritizing function is the execution frequency of the dependences. A simple approximation to the execution frequency of a dependence is the minimum of the execution frequencies of the producer and the consumer involved in the dependence. All the data dependences are ordered on the basis of the approximation. Going from the highest priority dependence to the lowest priority dependence, tasks are formed to include dependences, expanding any previously formed task that includes a higher priority dependence to include a lower priority dependence; if a previously formed task cannot be expanded to include a lower priority dependence without needing more than N successors, then the lower priority dependence remains exposed.

3.4.4 Special heuristics

Loop recurrences of any sort are treated as a special case for task selection. If a loop body with a recurrence is formed into a task, the recurrence causes the iterations of the loop to stall. The key problem is that the separation among the instructions involved in a recurrence decreases the overlap among successive iterations of the loop significantly leading to a substantial loss of performance. Figure 3-9 illustrates the impact of loop recurrences on performance. There are two options to avoid performance loss due to recurrences: (1) To schedule the instructions involved in recurrences so that the separation is closed statically or (2) to partition the loop body so that the separation is closed dynamically during execution. Scheduling techniques (described later in Section 4.4) may not succeed in statically closing the separation due to arbitrary control flow dependences and ambiguous data dependences. Partitioning the loop body may succeed better than scheduling by exploiting task-level hardware speculation support to overcome arbitrary dependences. Instead of incurring performance loss by selecting the entire loop body as a task, the recurrence (and hence the loop body) is partitioned into multiple tasks so that all the basic blocks in the recurrence are roots of their respective tasks. By partitioning the loop body, the recurrence, which is the critical path through the loop body, is executed without delay.

3.4.5 Short function invocations

As discussed before, basic blocks ending in function calls have been deemed terminal to avoid large tasks. This policy forces every function call to be partitioned as possibly many

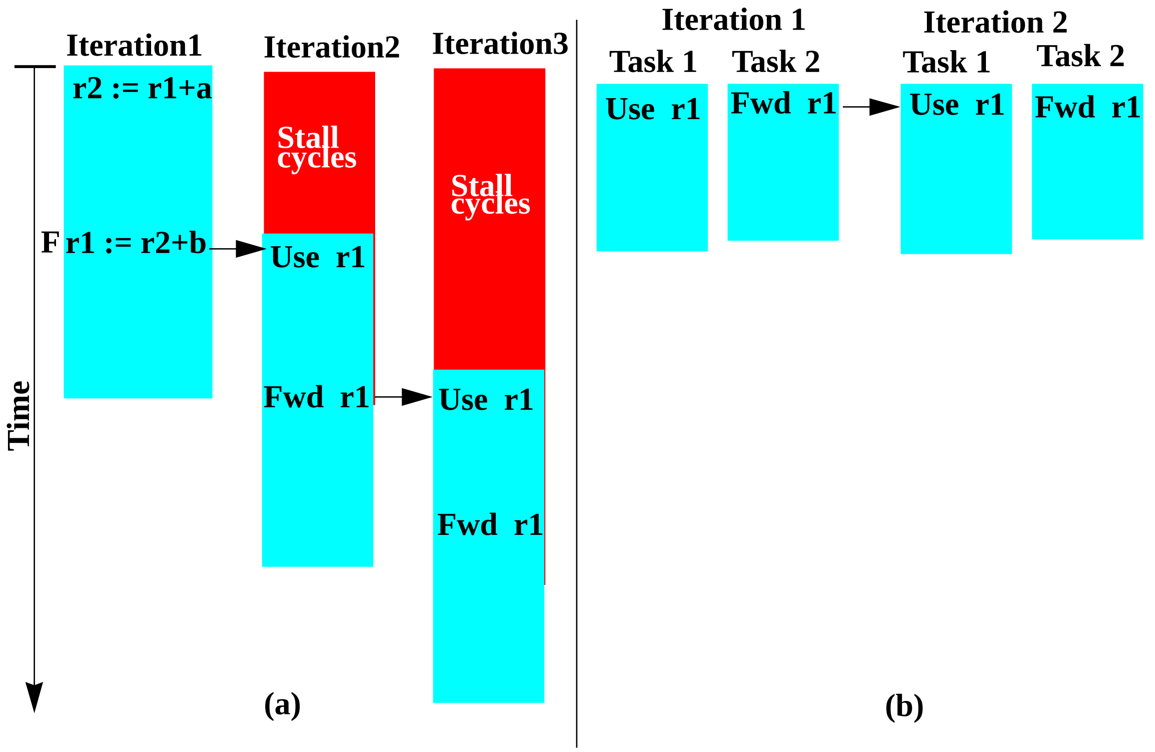


Figure 3-9: Effect of loop recurrences. (a) A loop body with a recurrence through register r1 is selected as a task resulting in the parallel execution of the iterations of the loop. F indicates forward of register r1. Use r1 denotes the instruction r2 := r1+a and Fwd r1 denotes the instruction r1 := r2+b. The arrows indicate data dependence. During execution, the loop iterations wait until the previous iteration produces and sends r1, leading to significant performance loss. (b) The loop body is partitioned into two tasks so that r1 is produced and sent before the next iteration needs it.

tasks. If, however, a function invocation executes only a few instructions then partitioning the few instructions into many tasks usually results in small tasks with many inter-task dependences. As a special case, these function invocations are included entirely within tasks. Function invocation that is included entirely within a task is called a **suppressed function call**.

Including entire function invocations raises additional concerns. Function definitions are partitioned into tasks and its instructions are annotated with Multiscalar task information (like register forward and task-exit bits) under the assumption that the hardware would execute these tasks in separate PUs. When a suppressed function call is executed, the tasks corresponding to the called function are executed in the same PU as the callee task, violating the assumption. So, execution of suppressed function calls may result in incorrect values of registers being sent, since the PUs are not executing subsequent tasks from the callee function but from the caller function. In addition to register forwards, task-exits of the tasks corresponding to the suppressed function call have to be ignored if all of those tasks are to be executed in one PU.

There are two ways of solving the problems with suppressed function calls: (1) Create two versions of those function definitions that are invoked through a suppressed function call; one version contains the usual Multiscalar annotations and the other is compiled without any Multiscalar annotations. The suppressed function calls use the non-Multiscalar version and the function calls that are not suppressed use the Multiscalar version. (2) Alternatively, correct execution can be achieved with a single version of function definitions using some hardware support. If a function call is suppressed, then hardware ignores any Multiscalar annotations as long as execution is in suppressed mode. Ignoring task-exit bits produces the desired effect of executing all the tasks of the suppressed function in one PU, and due to register-saving conventions used in function calls, ignoring register forwards produces the desired effect of forwarding register values that are visible after the return of the suppressed function call. The hardware easily identifies any function call as a suppressed function call if the task-exit bits of the call instruction are not set. If more function calls are made while executing within suppressed function calls, the entire chain of calls executes under suppressed mode. To track suppressed mode during nested suppressed function calls, the hardware matches function calls with returns by counting and as long as the count, maintained in the **suppress register**, is non-zero, the hardware continues to operate in suppressed mode. The suppress register is an implicit operand of every instruction and is an architectural register because it needs to be saved and restored during context switches and setjumps and longjumps.

With function call suppressing in place, I experimented with suppressing function calls that were in loops, so that entire loop bodies with function calls are demarcated as tasks. This strategy incurs significant performance loss due to memory dependences because the tasks contain several hundreds of instructions that modify global and heap variables. Memory dependence prediction and synchronization technique either stalls loads to be synchronized with stores or fails to prevent misspeculations due to the filling up of hardware tables used by the technique. Overall performance degrades considerably, making this approach inferior, compared to suppressing only short function invocations.

3.5 Implementation of task selection

Task selection grows tasks by starting with a single basic block and successively adding more basic blocks. The CFG is traversed visiting children nodes from parent node in the direction of control flow. As a node is visited, the task grows by including the node. At every edge of the CFG, a decision to include the current edge or not is made. If an edge is included, then the node it leads to is included in the current task and if the edge is exposed, then the node it leads to starts a new task. The process of growing a task simplifies to deciding, one at a time, whether or not to include a control flow edge. At every edge, the heuristics are consulted to reach a decision.

3.5.1 Implementing control flow heuristics

Figure 3-10 shows the pseudo-code implementation of the control flow heuristics discussed in Section 3.4.2. Task selection proceeds by maintaining two queues: *Task_q* and *explore_q*. *Task_q* contains the basic blocks that are the successors of terminal nodes and terminal edges of the tasks that have been explored. A new task is grown with each of the basic blocks contained in *task_q* as the root node. *Explore_q* contains the basic blocks that are members of the task being grown but have not been explored. Whenever a basic block is added to the task by putting on *task_q* to be explored further, the number of successors is checked and if it is fewer than the number of hardware targets, then the basic block is

```
task_selection() {
  add_to_task_q(basic_block_0);
  while (root = dequeue(task_q) {
      while (explore_q_is_not_empty)
          control_task(root, root);
  }
}

is_a_terminal_node(blk) {
  return (does_not_end_in_call(blk)
      && not_a_loop_end(blk)
      && not_a_loop_head(blk));
}
is_a_terminal_edge(blk, ch) {
  return(dfs_num(blk) < dfs_num(ch));
}
```

```
control_task(blk, root) {
   if (! is_a_terminal_node(blk) {
    for each child ch of blk {
       if (! is_a_terminal_edge(blk, ch)) {
           add_explore_q(ch);
           t = adjust_targets(root, blk, ch);
           if (t < TARGET_NUM) {
               feasible_task(root, blk, ch);
       } else {
           add_to_task_q(ch);
       }
    }
   } else {
       for each child ch of blk {
           add_to_task_q(ch);
       }
   }
}
```

Figure 3-10: Control flow heuristics. *task_selection()* is the top level driver. *control_task()* explores one basic block per invocation and queues the children of the basic block under consideration for further exploration. *control_task()* invokes *is_a_terminal_node()* and *is_a_terminal_edge()* which determine whether a node and an are terminal, respectively. *feasible_task()* tracks the basic blocks that correspond to fewer than TARGET_NUM successors.

added to the corresponding feasible task. Figure 3-10 shows the functions *is_a_terminal_node()* and *is_a_terminal_edge()*, which determine whether a given basic block or a control flow edge are terminal, respectively.

3.5.2 Implementing data dependence heuristics

Figure 3-11 shows the pseudo-code implementation of the control flow heuristics discussed in Section 3.4.3. To add data dependence heuristics to the task selection process described above, information about data dependences is obtained first. Producers and consumers corresponding to data dependences are identified by using standard compiler analyses. To include a data dependence edge from a producer basic block to a consumer basic

```
task_selection() {
  dep_list = sort_datadep_by_freq();
  for each (u,v) in dep_list {
    for each t = including_task of u {
        expand_task(u, t, (u,v));
        if (!include_dep_edge(t, (u,v))) {
        blk = early_codependent(t, (u,v));
        new_datadep_task(blk, (u,v));
    }
        for each (p,u) in CFG &&
        not_in_any_datadep_task(p,u) {
            new_datadep_task(u, (u,v));
        }
    }
    if (! datadep_task_rooted(blk_0)
        add_to_task_q(blk_0);
    while (root = dequeue(task_q) {
        if ( ! datadep_task_rooted(root)
            while (explore_q_not_empty)
                datadep_task(root, root, 0);
    }
}

expand_task(blk, task, dep_edge) {
    explore_q = task->explore_q;
    root = task->root;
    while (explore_q_not_empty &&
     ! include_dep_edge(root, dep_edge))
        datadep_task(blk, root, dep_edge);
}
```

```
datadep_task(blk, root, dep_edge) {
    if (! is_a_terminal_node(blk) {
     for each child ch of blk {
        if (! is_a_terminal_edge(blk, ch)
         && ! datadep_task_rooted(ch)) {
            if (codependent(ch, dep_edge))
                add_explore_q_head(ch);
            else
                add_explore_q_tail(ch);
            t = adjust_targets(root, blk, ch);
            if (t < TARGET_NUM) {
                feasible_task(root, blk, ch);
        } else {
            add_to_task_q(ch);
        }
     }
    } else {
        for each child ch of blk {
            add_to_task_q(ch);
        }
    }
}
new_datadep_task(blk, dep_edge) {
    root = add_to_task_q(blk);
    add_explore_q(blk);
    adjust_targets(root, blk);
    while (explore_q_not_empty) &&
     ! include_dep_edge(root, dep_edge))
        datadep_task(blk, root, dep_edge);
}
```

Figure 3-11: Data dependence heuristics. *task_selection()* is the top level driver. *datadep_task()* explores one basic block per invocation. Similar to *control_task(), datadep_task()* invokes *is_a_terminal_node()* and *is_a_terminal_edge()* to explore more basic blocks. codependent() determines whether a basic block is in the codependence set of a data dependence edge or not. *expand_task()* expands an existing task to include another data dependence edge. *datadep_task_rooted()* determines if a basic block is the root of a task or not.

block within a task, the dataflow equations given in Figure 3-8 are solved to identify the codependence set of the data dependence. The codependence set is used to guide the exploration of control flow paths by the control flow heuristics. If a data dependence edge starts from a basic block that is already included in a data dependence task, then the task is grown further to include the data dependence. Any basic blocks that are still left after data dependences are considered are explored using control flow heuristics.

When a data dependence is being included within a task, if a basic block that is not in the corresponding codependence set is included before all the basic blocks in the codependence set is explored and included, the extra basic block may introduce extra successors making the task infeasible; it may be the case that if just the codependence set is included before any other basic blocks, a feasible task is obtained. In order to ensure that the codependence set is included in the task before exploring other basic blocks during the inclusion of a data dependence in a task, *datadep_task()* inserts the basic blocks in the codependence set in *explore_q* ahead of other basic blocks. *Include_dep_edge()* determines if the codependence set is included within a feasible task; if so, the task is not grown any further, so that basic blocks do not get included within the task needlessly causing other data dependences to be exposed. If a data dependence cannot be included within a task due to exceeding the number of hardware targets, one of the following actions is taken: (1) if the producer is not already in a task, then a task is grown with the producer basic block as the root, or (2) if the producer is already in a task, then a new task is grown so that the consumer is placed late in the new task. The first case is straightforward but in the second case, the compiler has to determine the basic block to be used as the root of the new task, which is grown to include the consumer. *Early_codependent()* determines the basic block with the smallest depth-first-search number that is in the codependence set of the data dependence but not included within the task in which the producer is included. This basic block is used as the root to grow a task in which the consumer is included. This strategy facilitates overlap of computation with communication of values corresponding to the data dependences that were exposed, as discussed in Section 3.3.3.

3.5.3 Code replication

Since Multiscalar tasks can be entered at only one point, basic blocks that are common between two tasks have to be replicated. Figure 3-12 illustrates code replication in task selection. This replication is similar to tail-duplication of superblocks [48] and trace

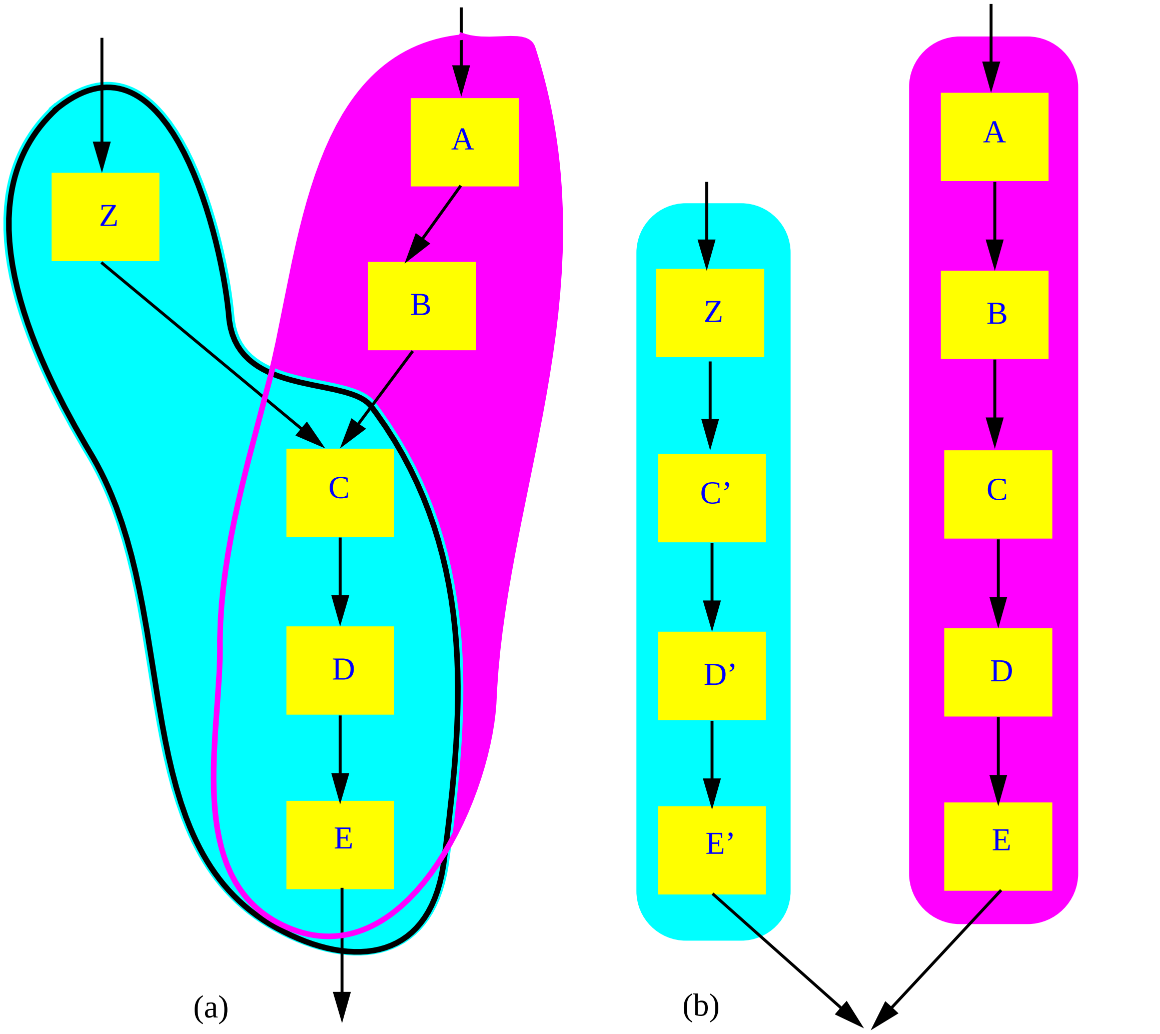


Figure 3-12: Code replication during task selection. (a) A part of the CFG of a program. Shaded regions represent tasks. Task selection partitions the CFG into two tasks such that basic blocks C, D, and E. (b) In the actual code layout, the basic blocks C, D, and E are replicated into C', D', and E', respectively in one of the two tasks.

scheduling [33] [34]. If two control flow edges converge to a basic block and one of them is included in one task and the other in another task, then the basic block and successor basic blocks are replicated in each of the tasks.

3.5.4 Code layout

After task selection is done, the intermediate representation of the code is laid out linearly so that code segment corresponding to each task is contiguous. Tasks are ordered linearly based on the depth-first-search number of the roots of each task. Within each task, the basic blocks are ordered linearly based on their depth-first-search number. It is possible that for a branch, the taken edge may be included in the task containing the branch and not the fall-through edge; when the code for the task is laid out, the taken path is laid out following the branch and the sense of the branch condition is reversed so that no extra jumps are added due to the new code layout. Figure 3-13 illustrates the problem of code layout after task selection.

3.6 Related work

There has been a myriad of compiler techniques developed to automatically discover parallelism in non-numeric programs written in imperative languages. As far as discovery of parallelism is concerned, the problem remains the same but support for speculation in the Multiscalar architecture changes the extent to which the compiler may be aggressive and thereby, presents hitherto unexplored opportunities.

Previous work on partitioning functional language programs to execute on multiprocessors differs from this work in many ways. Since the programs considered were functional, considerations of data dependencies were much simpler. The algorithms considered tasks at the granularity of entire function definitions and not at the fine granularity of a few basic blocks, as is the case for Multiscalar tasks. Sarkar [89] showed that the problem of opti-

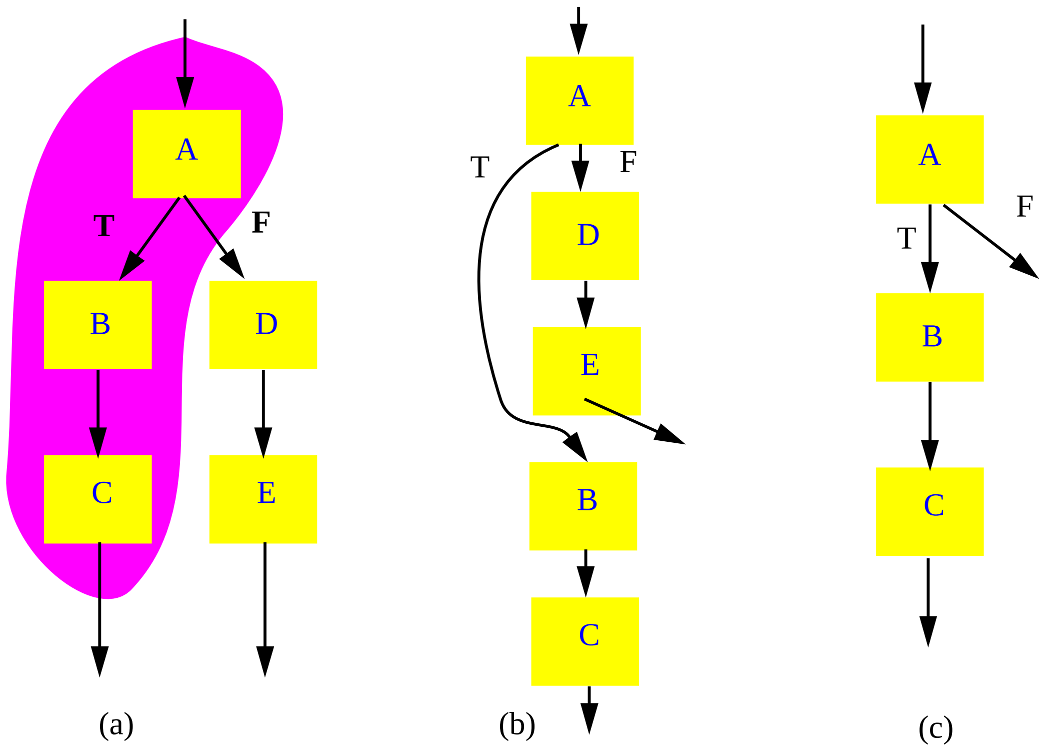


Figure 3-13: Code layout after task selection. (a) A part of the CFG of a program. The shaded region represents a task. The taken edge (A,B) is included in the task and the not-taken edge (A,D) is not included in the same task. (b) The original layout of the code. (c) Code layout for the task. The sense of the branch at the end of basic block A is reversed.

mally partitioning simple functional programs to execute on a Multiprocessor considering only the amount of work and data communication is NP-Complete.

In the past, several compilation structures, for example, trace in Trace Scheduling [33] [34], superblock [48] and hyperblock [67] in IMPACT [22], region in region scheduling [2] [45] have been proposed to facilitate the development of many scheduling techniques. Multiscalar tasks differ from these structures fundamentally in that tasks are execution structures whereas the others are compilation structures. Since different tasks get executed on different processing units, inter-task control and data independence is of primary importance. Inter-structure dependences between different traces, superblocks, hyperblocks and regions are of little significance. Since all these structures are used in global scheduling, intra-structure dependencies are the only significant dependences. Since tasks are execution structures intimately connected to the prediction mechanism, they may not have arbitrarily many number of successors. The other structures do not have any such constraints. The criteria used to select the other structures are usually execution frequency of control flow paths, type of control flow (for e.g., traces stop at backward branches), presence of function calls, etc. The criteria used to select tasks are number of successors, data dependences crossing task boundaries, control flow among tasks (entire loop bodies are better suited to be single tasks rather than multiple tasks because of intra-iteration dependences) and the number of instructions.

Much of the work on automatic parallelization [6] [9] [10] [63] [65] [110] [111] [112] [116] and vectorization [4] [5] [14] of Fortran programs is related to Multiscalar compilation. While automatic parallelization attempts to partition numeric code so that loop iterations in parallel, most of Multiscalar task selection attempts to partition mainly non-numeric code with complex and ambiguous control and data dependencies. Most of the automatic parallelization involves accurate detection of memory independencies, in numerical programs with complex array indices and more straightforward control dependences; where as most of Multiscalar task selection involves task selection in the presence of ambiguous dependences. Moreover, automatic parallelization did not have much of the architectural support like speculative execution, register and memory renaming and dynamic memory disambiguation that the Multiscalar architecture provides. Hence, many loops that would not be considered for automatic parallelization are speculatively executed in parallel in the Multiscalar scenario.

Chapter 4

Register Communication

As tasks execute, data values are produced and consumed within the same task and among different tasks, corresponding to intra-task and inter-task communication, respectively. These data values are bound to memory and register storage locations. In the case of memory storage, it is difficult to determine precisely the producers and consumers of data values since memory storage names are determined dynamically (via address calculations). On the other hand, in the case of register storage, it is straightforward to identify producers and consumers since all register storage names are known statically (via register specifiers).

Regardless of the type of storage involved, data values passed between instructions represent a critical factor, impacting two key aspects of program execution: correctness and performance. To ensure maintaining correctness, data values must be communicated from producing to consuming instructions as dictated by program semantics. To avoid constraining performance, data values must be communicated from producing to consuming instructions as soon as possible. Inter-task communication of data values impacts overall performance because critical paths through tasks typically involve computation that consumes register values from predecessor tasks and produces values for successor tasks and performance may suffer if these critical paths are aggravated unnecessarily.

In this chapter, I focus on the communication of data values bound to register storage for two reasons (both related to the prevalence of the load-store model of computation). First, this type of communication is the most common. Second, it is the most amenable to analysis. We investigate a number of alternative register communication strategies for the Multiscalar architecture. In Section 4.1, I present the Multiscalar register model and correctness criteria. In Section 4.2, I describe a range of alternative register communication strategies. I begin with a simple base strategy that guarantees correctness; I progressively incorporate compiler strategies to achieve higher performance [106]. In Section 4.3, I discuss the implications of dead registers and compiler register assignment. All of the strategies vary the timing of communication but do not move the computation that generates the values involved in the communication. Extending this analysis further, I describe scheduling techniques to mitigate register communication delay [105], in Section 4.4. In Section 4.5, I describe the details of the implementation of register communication generation. In Section 4.6, I discuss related work.

4.1 Register communication model

The Multiscalar architecture provides a distributed physical register file implementation of a single logical register file [18]. I now explain the abstract model of communication of register values among the different physical register files to maintain the semantics of a single logical register file.

In the set of all architectural registers, there are two mutually exclusive and collectively exhaustive subsets: (1) The set of registers that may be modified in the current task, called the **ModSet** and (2) the set of registers that are guaranteed to be not modified in the task, called the **UnModSet**. During execution, every task eventually receives values for all the architectural registers from its predecessor and eventually sends values for all the architectural registers to its successors[1]. Also, assume that each task operates on its own set of hardware registers. The Multiscalar architecture supports this assumption by providing each PU with its own physical register file. When a register value arrives at a task, the hardware identifies the register either as a ModSet register or as a UnModset register, by consulting the ModSet. The task binds the register value to the hardware register of the PU, regardless of whether it is a ModSet register or a UnModSet register. Any register value generated during the course of executing the instructions of the task is also bound to its appropriate hardware register. The distinction between a ModSet and a UnModSet register is important only when the task sends the register value to successor tasks. For a UnModSet register, the same value that was received, is propagated to successor tasks. But for a ModSet register, the value that was received, is stopped from propagating further; the value bound to the register after the last modification of the register is done in the task is sent. Figure 4-1 illustrates the Multiscalar register communication model.

The ModSet may be determined by some hardware or the compiler. If tasks are large and span several basic blocks and include arbitrary control flow, then the compiler can assist hardware by providing the ModSet on a per task basis. The create mask described in Chapter 2 is a bit vector representation of the ModSet. (The UnModSet is the complement of the ModSet and need not be provided separately). In addition to determining the ModSet for each task, the compiler or hardware also has to identify where in the task each ModSet register may be sent to successor tasks. This is the point in the task beyond which the register is guaranteed not to be modified by the instructions of the task.

The problem of identifying the ModSet is simple and straightforward; it is the union of the sets of registers that may get modified in any path through the task. Since tasks may have complex control flow within them, the problem of identifying which value of a ModSet register to send is more involved. There are two correctness constraints: (1) every path

[1] In an actual implementation, many hardware optimizations may reduce the bandwidth demands of register values. By doing some bookkeeping, in hardware, the UnModSet registers need not be propagated repeatedly if they have already been sent to all PUs.

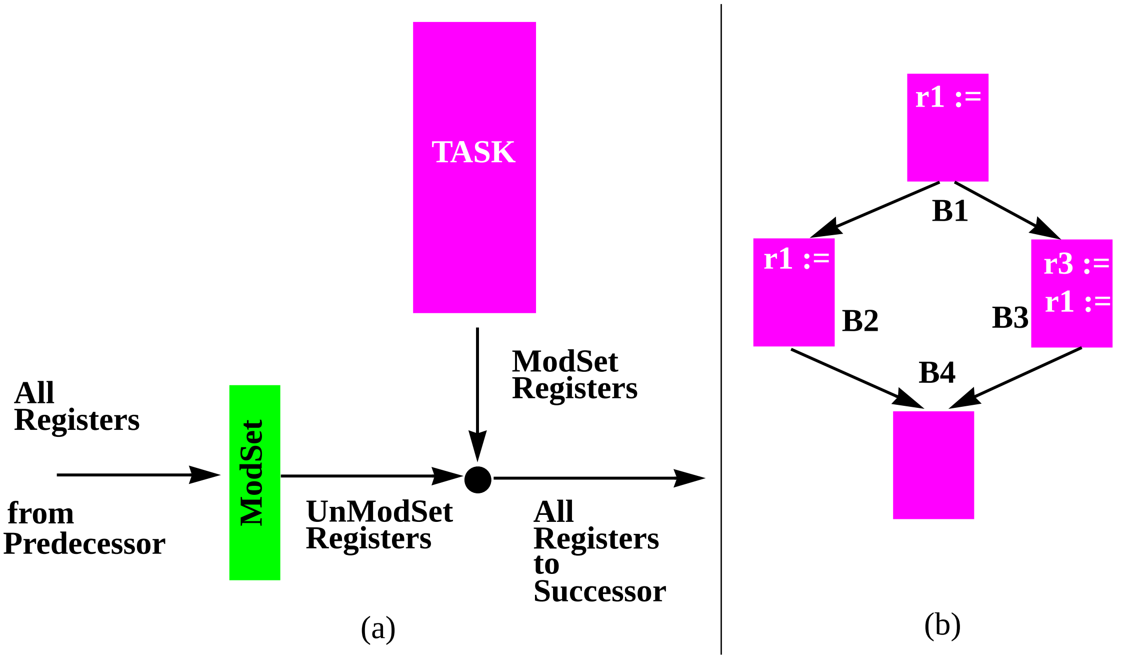


Figure 4-1: Register communication model. (a) Register communication abstraction. All of the registers arriving at a task are filtered by the ModSet so that the old values of the UnModSet registers are propagated and the new values of the Modset registers are sent, as and when generated by the task. (b) Register communication correctness criteria. The task comprises basic blocks B1, B2, B3 and B4. The task modifies registers r1 and r3. Depending on the path taken through the task, value of r1 corresponding to B2 or B3 must be sent; value of r3 corresponding to B3 or propagated from the predecessor task must be sent.

through the task must send all the registers in the ModSet and (2) for each register in the ModSet, the value corresponding to the basic blocks beyond which the register remains unmodified, irrespective of the path taken through the task, must be sent. The first constraint ensures that successor tasks do not get starved for a register, irrespective of the path taken through the task and the second constraint guarantees that each task sends its successor tasks the last updated register values, which are the correct values as per sequential semantics of the program. It is important to note that the second constraint does not preclude sending of multiple speculative values of a register, but the last value sent must be the correct value as per sequential semantics. Any task that receives a register value that it has already received may squash the computation dependent on the register and recompute with the latter value. Such squashes are the result of register data speculation. Together, the two constraints guarantee forward progress and correctness.

Figure 4-1(b) illustrates the two correctness criteria through an example.The example shows a task with four basic blocks B1, B2, B3, and B4. Registers r1 and r3 are modified by the task, so the ModSet contains r1 and r3. There are two paths through the task: B1B2B4 and B1B3B4. The first criterion states that both r1 and r3 must be sent from either paths; otherwise, successor tasks will deadlock if they need the value of either r1 or r3. Applying the second criterion to r3 is simple because it is modified only in B3. If the path B1B3B4 is taken, then the value of r3 after it is modified in B3 must be sent; otherwise, the value of r3 received from the predecessor task must be propagated to successor tasks. Applying the second criterion to r1 is more interesting than r3. If the path B1B3B4 is taken, then the value of r1 after it is modified in B3 must be sent; otherwise, the value of r1 after it is modified in B2 must be sent. It is possible to send the value of r3 received from the predecessor task before the branch in B1 is resolved speculating that the path B1B2B4 will be taken by the task, but if the path B1B3B4 is taken, then the updated value of r3 must be sent.

The correctness criteria identify the register values that need to be sent to successor tasks to maintain sequential semantics. Performance criteria influence when the correct register values may be sent. In the example of Figure 4-1(b), values of r1 and r3 may be sent at the end of the task. If successor tasks use the values then they stall until the values are received. In order to avoid stalling consumer tasks for register values, it is crucial that register values are sent as soon as possible.

4.2 Strategies for orchestrating register communication

In this section, I consider a progression of four strategies for register communication to achieve high performance. The four strategies are **end_send**, **eager_send**, **last_send**, **and**

spec_send. The strategies differ in how early the ModSet register values are sent to successor tasks. Before going into the details of the various strategies, I illustrate them through an example. Consider the task in Figure 4-2 comprising basic blocks B1, B2, B3 and B4. Let us assume that the edge B1B2 is taken with probability 0.9 and the edge B1B3 is taken with probability 0.1.

The simplest strategy which ensures correctness, called end_send, is to send the values of all the ModSet registers at the end of the task. In end_send (Figure 4-2(a)), both r1 and r3 are sent at the end causing subsequent tasks that need r1 or r3 to wait.

A possible improvement in performance over this simple strategy is to send the value of a ModSet register every time it is modified. This strategy, called eager_send, sends ModSet register values as soon as and every time they are modified, causing computation in successor tasks that use all but the last value to be squashed and restarted. Since the value corresponding to the last modification of each register is sent last[2], this strategy preserves the semantics of the program. Although eager_send may send register values earlier than end_send, eager_send may send the same register multiple times, resulting in squashing of subsequent tasks. Eager_send (Figure 4-2(b)) sends r1 as soon as it is defined in B1. Since there is a define of r1 in both B2 and B3, r1 is sent again, causing a squash. If B3 is taken then r3 is sent as soon as it is defined, otherwise it is sent at the end. Thus, eager_send sends r3 earlier than end_send if B3 is taken but incurs extra squashes due to sending incorrect values of r1. In last_send (Figure 4-2(c)), both r1 and r3 are sent from B2 or B3, since they are the last modifications of the registers. Since B2 does not define r3, an extra instruction may be inserted to send r3.

If the last modification of each ModSet register is known, however, multiple sends of the same register (and the resultant squashes) can be avoided. To this end, last_send sends

[2] The hardware is responsible for preserving the order among multiple sends of the values of the same register.

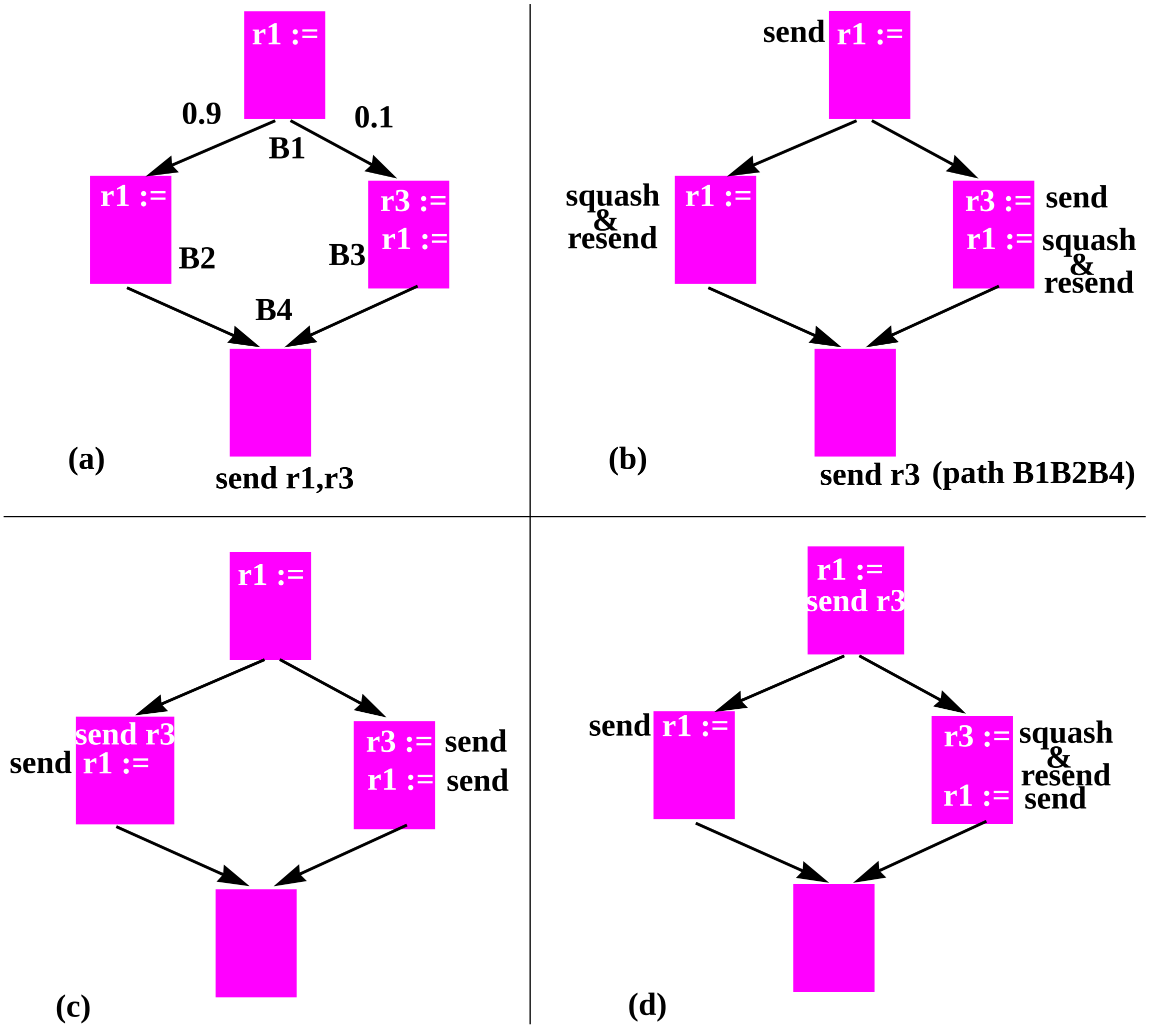


Figure 4-2: An example illustrating the four strategies. The task comprises basic blocks B1, B2, B3 and B4. The edge B1B2 is taken with probability 0.9 and the edge B1B3 is taken with probability 0.1. The ModSet for this task contains registers r1 and r3. (a) End_send: Both r1 and r3 are sent at the end. (b) Eager_send: r1 is sent in B1 but since there is a define of r1 in B2 and B3, r1 causes a squash and is sent again. If B2 is taken then r3 is sent at the end otherwise r3 is sent in B3. (c) Last_send: r1 and r3 are sent from B2 and B3. Sending r3 from B2 may require an extra instruction and in the other cases the existing instructions may be annotated to send their destination registers. (d) Spec_send: Since B2 is more frequent than B3, r3 is speculatively sent from B1 and r1 is sent from B2 and B3 like last_send. If B3 is taken then r3 causes a squash and is sent again. Sending r3 from B1 may require an extra instruction and in the other cases the existing instructions annotated like last_send.

ModSet register values as soon as there is a guarantee that the register will not be modified again. the compiler identifies the last modification of each ModSet register and marks it explicitly for communication. Although last_send avoids squashes by sending only after the last modification of the registers, in the presence of control flow, it is conservative and may delay sending register values. Last_send avoids the squashes incurred by eager_send by sending only the correct values of r1 and sends r3 earlier than eager_send via an explicit instruction.

The last strategy, known as spec_send is a more controlled version of eager_send in that it sends ModSet register values as soon as there is a high probability that the register will not be modified again. spec_send speculatively sends register values corresponding to last modifications from “high frequency” paths and resorts to squashes if any “low frequency” paths modify a previously sent register. Spec_send attempts to avoid both delaying register values due to infrequent paths and also excessive squashing due to incorrect speculation. Spec_send (Figure 4-2(d)) speculatively sends r3 from B1, since B3 is infrequently taken and the frequent path of B1B2B4 does not define r3. If B3 is taken then r3 is sent again, causing a squash. r1 is sent from B2 or B3 like last_send. Spec_send accelerates r3 even more than last_send by taking advantage of speculation. In the frequent case of B2 being taken, r3 is sent earlier from B1 via an explicit instruction and in the infrequent case of B3, r3 causes a squash.

4.2.1 End_send and eager_send

I now describe the issues in the realization of each of these four strategies. Both end_send and eager_send strategies may be implemented with relatively simple hardware. These two strategies require no explicit send instructions either, implying little ISA change. Since the Multiscalar architecture employs both memory data speculation and control speculation, hardware to squash incorrect speculation in eager_send is already present. Therefore, squashes due to multiple register sends can be performed without any additional hardware. On one hand, this scheme sends the register values earlier than the previous scheme, but on the other, a register may get sent too early speculatively and may result in a squash. There is a trade-off between avoiding communication stalls by sending a register value early speculatively and incurring squashing loss due to incorrect register speculation. The compiler is employed to perform this trade-off in spec_send.

Last_send and spec_send may require compiler analyses to determine the last modification of each register, in the presence of complex control flow. Moreover, hardware or ISA support as described in Section 2.4, may be needed to efficiently convey information from the compiler to the hardware. The analyses required to generate register communication may be formulated in terms of existing dataflow frameworks. I present the dataflow equations to explain how the compiler determines and generates the required register communication. In all the following discussion of dataflow equations, it is assumed that if there are multiple modifications of a register within a basic block, then the last modification is marked for communication.

4.2.2 Last_send

The key issue in both last_send and spec_send is to determine the basic blocks after which each ModSet register is guaranteed to be not modified in the presence of complex control flow in the task and to generate sends for the register in the earliest such basic block down any path through the task. *Nomoredef* identifies the set of the basic blocks after which a register is guaranteed not to be modified any further in each task. Send identifies the earliest basic block down any path through the task among such basic blocks. Figure 4-3 lists the dataflow equations for last_send.

Nomoredef(i,r) is true if beyond basic block i, register r is not modified in the current task. *BBdef(i,r)* is true if register r is modified in basic block i. The equation asserts that for every child of basic block i, if there is no modification of a register beyond the child and

$$\mathrm{Nomoredef}(i, r) = \prod_{j \in \mathrm{children}(i)} \{\mathrm{Nomoredef}(j, r) \cap \overline{\mathrm{BBdef}(j, r)}\}$$

$$\mathrm{Send}(i, r) = \left\{ \sum_{j \in \mathrm{parents}(i)} \overline{\mathrm{Nomoredef}(j, r)} \right\} \cap \mathrm{Nomoredef}(i, r)$$

INITIAL VALUES: Nomoredef(i,r) = TRUE

Figure 4-3: Dataflow equations for last_send. *BBdef(i,r)* is true if register *r* is modified in basic block *i*. *Nomoredef(i,r)* is true if there is a guarantee that register *r* is not modified in any successor of basic block *i* in the current task. *Send(i,r)* is true if *Nomoredef(i,r)* is true and register *r* has not been sent in at least one path from the root of the task to basic block *i*.

the child does not modify the register, then there is no modification of the register beyond basic block i. A key property of *Nomoredef* is that if *Nomoredef* is true for a basic block, then it is true for all its successors in the task.

The second equation in Figure 4-3 computes the vector *Send*. *Send(i,r)* is true if the register r is not sent in at least one path from the root of the task to basic block i and *Nomoredef(i,r)* is true. The equation asserts that if there is no modification of a register beyond basic block i and for at least one of the parents of basic block i, *Nomoredef* is false, then the register is sent from basic block i. Send identifies the basic block in any path from the root of the task, at which *Nomoredef* changes from false to true. Since *Nomoredef* is true for all the successors of the basic block (due to the property stated above), that is the earliest basic block in the path at which the register may be sent. Note that in a path from the root of the task, there may be multiple basic blocks for which *Nomoredef* changes from false to true; therefore, *Send* may be true for multiple basic blocks in a path. The hardware ignores any sends of a register encountered after a send for the same register.

4.2.3 Spec_send

With the analysis of last_send, a register is not sent until it is guaranteed that there are no more modifications to the register. This guarantee implies that if an infrequent basic block that modifies a register much later in the task, then the register is delayed until much later in the task. Frequency information used by this analysis may be obtained from static or dynamic profiling. If the register is sent earlier in the task, however, and a special “resend” of the register is done in the infrequent basic block, on an average, the register is not delayed as before. In the infrequent case of the basic block being executed, the resend of the register causes a squash and restart with the correct value but in the frequent case of the basic block not being executed, the register is sent much earlier and no squash is suffered. Figure 4-4 lists the equations for spec_send. The key difference between the analysis for last_send and spec_send is that spec_send ignores the “low frequency” defines of registers and computes *Spnomoredef* and *Spsend* exactly like *Nomoredef* and *Send*. *Validatesend* determines the extra “resends” corresponding to the “low frequency” defines ignored in *Spsend*.

Spnomoredef(i,r) is true if beyond basic block i, register r is not modified in any “high frequency” basic block in the current task. *Hifreqdef*(i,r) is true if register r is modified in basic block i and the execution frequency of the basic block is greater than a fraction (*SpThreshold*) of the frequency of the root of its task. The equation asserts that for every child of basic block i, if there is no “high frequency” modification of a register beyond the child and the child does not “high frequency” modify the register, then there is no “high frequency” modification of the register beyond basic block i. *Validatesend(i,r)* is true if i is a “low frequency” basic block which modifies r and *Spnomoredef(i,r)* is true. *Validatesend* computes extra resends of registers that were ignored due to their “low frequency”. Note that *Validatesend* may cause multiple resends of the same register resulting in multiple squashes. Such multiple squashes can be avoided by constraining *Validatesend* more.

$$\mathrm{Spnomodef}(i, r) = \prod_{j \in \mathrm{children}(i)} \{\mathrm{Spnomodef}(j, r) \cap \overline{\mathrm{Hifreqdef}(j, r)}\}$$

$$\mathrm{Spsend}(i, r) = \left\{ \sum_{j \in \mathrm{parents}(i)} \overline{\mathrm{Spnomoredef}(j, r)} \right\} \cap \mathrm{Spnomodef}(i, r)$$

$$\mathrm{Hifreqdef}(i, r) = \mathrm{BBdef}(i, r) \cap \mathrm{Hifrnode}(i)$$

$$\mathrm{Hifrnode}(i) = (\mathrm{freq}(i)) / (\mathrm{freq}(\mathrm{root})) > \mathrm{SpThreshold}$$

$$\mathrm{Validatesend}(i, r) = \mathrm{Spnomoredef}(i, r) \cap \mathrm{BBdef}(i, r) \cap \overline{\mathrm{Hifrnode}(i)}$$

INITIAL VALUES: $\mathrm{Spnomoredef}(i, r) = \mathrm{TRUE}$

Figure 4-4: Dataflow equations for spec_send. *Hifreqdef(i,r)* is true if register *r* is modified in basic block *i* and the execution frequency of the basic block is greater than a fraction (*SpThreshold*) of the frequency of the root of its task. *Spnomoredef(i,r)* is true if there is a guarantee that register *r* is not modified in any “high frequency” successor of basic block *i* in the current task. *Spsend(i,r)* is true if *Spnomoredef(i,r)* is true and register *r* has not been sent in at least one path from the root of the task to basic block *i*. *Validatedef(i,r)* is true if *i* is a “low frequency” basic block that modifies the register *r*.

The compiler solves the system of dataflow equations to determine the basic blocks for which *Send* or *Spsend* and *Validatesend* are true. If such a basic block contains an instruction that modifies the register, then the instruction is annotated to forward the register and otherwise, an extra release instruction (introduced in Section 2.4.2) is inserted at the top of the basic block to send the register value. Since register communication is fairly dense, avoiding extra release instructions through annotation of existing instructions is imperative. In Figure 4-5, there are two examples of tasks with register communication annotations using last_send.

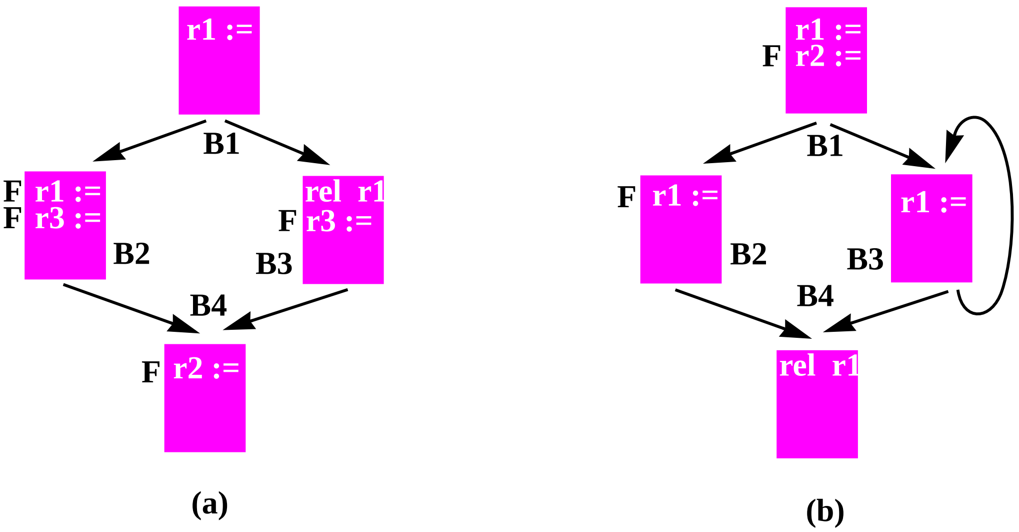


Figure 4-5: Register communication annotation. Two examples of tasks with register communication annotations are shown. Forwards of registers are indicated by F's beside the instructions. In example (a), in the path B1B2B4, r1 is last modified in B2 and is forwarded from B2 and in the path B1B3B4, r1 is not modified past B1 and is released in B3. r3 is last modified in B2 and B3, in paths B1B3B4 and B1B2B4 respectively and is forwarded from them. r2 is modified only in B4 and is forwarded from there. In example (b), r1 is last modified in B2 in the path B1B2B4 and is forwarded from B2 since there is a loop in the path B1B3B4, r1 is guaranteed to not change only on exit of the loop and so is released in B4.

4.3 Implications of dead registers and register assignment

Irrespective of the register communication strategy used, dead register information can be used to improve register communication and register assignment may unnecessarily disallow sending of a register value early in the task.

4.3.1 Dead registers

A register is said to be dead at a basic block if its value is not used beyond the basic block [1]. If a register is dead beyond a task, then the register value need not be sent to successor tasks. Dead register information is used to reduce register communication bandwidth demand and avoid the overhead of extra release instructions as well. Register communication bandwidth on the register communication interconnect (Section 2.5) is an important resource which if over-burdened may lead to queuing up of register values aggravating performance loss due to register communication delays. In Figure 4-5(a), if r1 is dead beyond B4, i.e., the value of r1 is not used by any successor of B4, then r1 need not be sent to successor tasks, saving register communication bandwidth and avoiding the extra release instruction in B3.

Dead register information may be conveyed to the hardware by providing an explicit dead register mask in the task descriptor, similar to create mask. Registers in the dead register mask are neither forwarded nor released. Since the register is dead, no successor task waits for the register value and so, not propagating the register does not cause any starvation. All four register communication strategies can take advantage of the dead register mask.

The dead register mask adds to the overhead of static code size increase due to task descriptors. Alternatively, dead register information may be provided by overloading the ModSet. Registers that are dead beyond a task are included in the ModSet of the task, regardless of whether they are defined in the task or not; but no forward or release is done. If the register does get defined in the task, then the value is used within the task and does not get propagated to successor tasks. If the register does not get defined in the task, then the value from a predecessor task is used within the task but does not get propagated to successor tasks. Only last_send and spec_send can take advantage of ModSet overloaded with register dead information. end_send and eager_send cannot distinguish between dead ModSet registers and live ModSet registers since they do not use any specification to identify register value sends.

Another issue with taking advantage of dead register information is debugging. Dead register optimization may cause register-allocated variables to hold a value that does not correspond to the last modification of the register. The value of a dead register is not propagated beyond the task beyond which the register is dead; if the register is inspected during debugging at a program point beyond the task, the register may hold a bogus value. But this problem exists for dead register optimization used by register allocators for superscalar processors. The solutions adopted by superscalar architectures may be used, which is to turn off dead register optimizations during debugging.

4.3.2 Register assignment

Anti-dependencies introduced by register assignment may delay forward of a register value in any of the strategies. If register assignment uses the same register name for two live ranges that co-exist in mutually exclusive[3] (like the if-path and the else-path of a branch) control flow paths, then the value bound to the register name cannot be forwarded until the control flow is resolved. Figure 4-6 illustrates the problem through an example. Figure 4-6(a) shows a task with four basic blocks. Pseudo-register r1 is dead in B4 but is live out of B3. Pseudo-register r2 is live out of B4. Pseudo-registers r1 and r2 have non-overlapping live ranges and hence, can be assigned the same physical register. Figure 4-6(b) shows such a register assignment. Assuming last_send, this assignment causes value of $1 to be sent only in B3. Round-robin assignment of registers may mitigate the problem. Under last_send, a round-robin assignment, as shown in Figure 4-6(c), avoids delaying the forward of $1 by assigning another register to r2. Under eager_send and spec_send, if $1 value is sent from B1 and the path B2B4 is taken, then squashes ensue for the assignment in Figure 4-6(b) but the round-robin assignment, as shown in Figure 4-6(c), would not incur any such squashes.

[3] Note that the same register name cannot be used for live ranges that co-exist on control flow paths that are not mutually exclusive; such a register assignment violates program semantics.

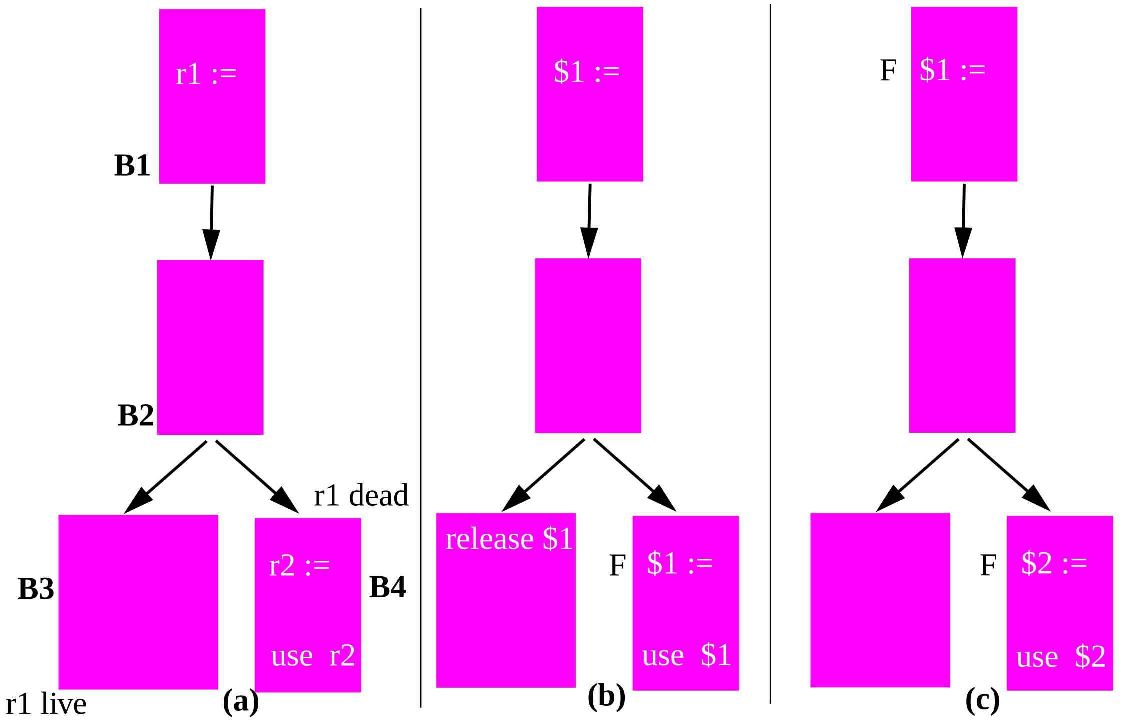


Figure 4-6: An example of register assignment delaying register forward. (a) A task with 4 basic blocks before register allocation. r1 and r2 are pseudo-registers. r1 is dead at the beginning of B4 but live at the end of B3. (b) Both r1 and r2 are assigned the same register $1, resulting in the release of $1 in B3. (c) r1 is assigned $1 and r2 is assigned $2. $1 is forwarded in B1, much earlier than B3.

4.4 Register communication scheduling

In Section 4.2, I explained how register value communication is identified and specified to the hardware. All of these strategies vary the timing of communication but do not move the computation that generates the values involved in the communication. Extending this analysis further, I have devised and implemented a static scheduler that moves computation to hide inter-task register communication delay further. In this section, I consider how the compiler can schedule the communication to improve performance.

Critical paths through tasks typically involve dependence chains that consume register values produced by predecessor tasks and produce register values to be consumed by successor tasks. An unnecessary aggravation of these critical paths can result in performance loss, as illustrated in Figure 4-7. This loss can be mitigated by alleviating the unnecessary aggravation of the critical path, i.e., by sending the register value early during the execution of the producer task and consuming the register value late during the execution of the consumer task. If a register value has to be sent early, the instruction that produces the value (called the **producer instruction**) has to be executed early. The instruction may be executed early, if it is statically scheduled to be as close to the beginning of its task as pos-

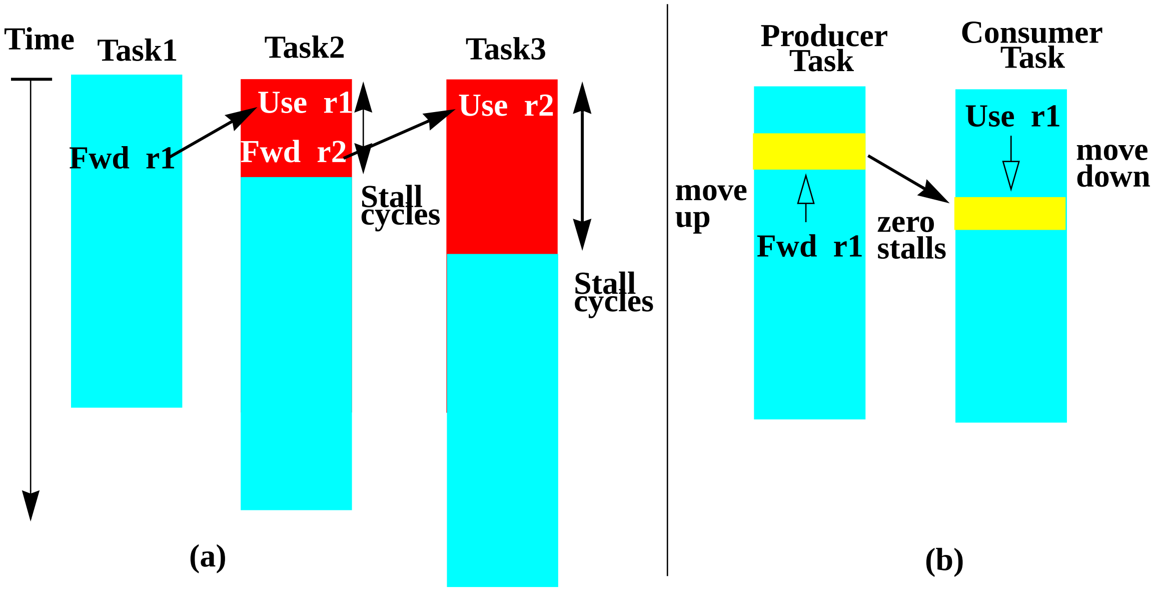


Figure 4-7: Register communication dynamics. a) Performance loss due to register communication. The forward of r1 in Task1 causes the use of r1 in Task2 to stall. Fwd r2 is dependent on Use r1, which causes the forward of r2 to be delayed. The delays cascade causing Task3 to stall for r2. (b) Scheduling register communication. The forward of r1 is moved up to be done early and the use of r1 is down to be done late so that stalls are reduced.

sible[4]. Similarly, a register value may be consumed late, if the instruction that consumes the value (called the **consumer instruction**) is statically scheduled to be as close to the end of its task as possible.

Figure 4-8 illustrates the above concepts using the run time dynamics of an example. In the example, Task1 produces the value for the variable *tmp* late which causes Task2 to be stalled because it needs the value. Moving the instruction that produces *tmp* up in Task1 causes Task1 to send the value early, which enables Task2 to proceed without stalling.

Moving a producer instruction of a task up in the static code may require moving the instructions (within the task) it depends on (i.e., the backward slice [52] [108] of the producer instruction within the task), up. Similarly, moving a consumer instruction of a task down may require moving the instructions (within the task) that are dependent on it (i.e., the forward slice [108] of the consumer instruction within the task) down. Typically, critical paths through tasks constitute dependence chains that start with a consumer instruction and end with a producer instruction. A complication that arises is that the instructions on such dependence chains need to be moved both up and down because on one hand, they are on the backward slice of the producer and on the other, they are on the forward slice of the consumer. The decision to move such instructions up or down depends on whether the instructions cause more stalls due to production or consumption.

4.4.1 Scheduling strategy

The main components of the scheduling algorithm: (1) identification of which instructions to move in which direction, (2) transformations to perform code motion of the instructions in the appropriate direction, and (3) estimation of the stall each instruction incurs due to register communication through a cost model.

[4] Even for out-of-order issue engines, if instruction I_i is statically placed before instruction I_j in program order, then instruction I_i is dispatched before instruction I_j.

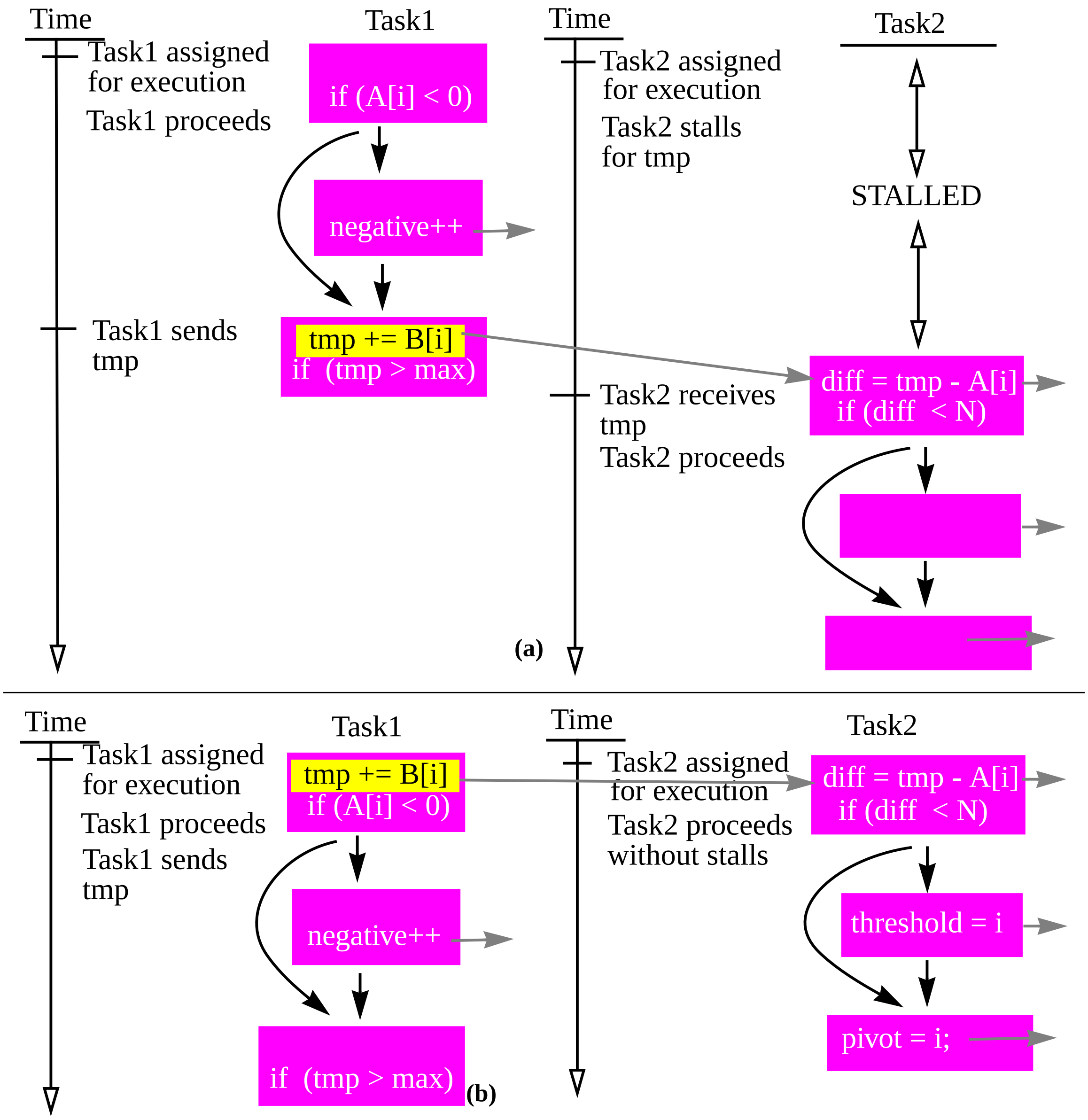


Figure 4-8: Run time dynamics of the example. (a) In the example, task2 is stalled for the register value tmp because task1, which is the producer, produces the value late. (b) In task1, the instruction that produces tmp is moved up. In task2, the instruction that consumes tmp cannot be moved down. Nevertheless, task1 sends tmp early and task2 proceeds without stalling.

4.4.2 Identifying the instructions for scheduling

The scheduler identifies two categories of instructions. In the Multiscalar architecture, the instructions that produce register values for other tasks are called forwarding instructions (as defined in Section 2.4.2). The instructions that consume register values from the forwarding instructions of prior tasks are called **receiving instructions**. All forwarding instructions and their backward slices are categorized as **sourcing instructions**. All sourcing instructions are to be moved up in the static code. All receiving instructions and their forward slices are categorized as **sinking instructions.** All sinking instructions are to be moved down in the static code.

If an instruction is both a sourcing and a sinking instruction, then depending on whether it causes a larger delay as a source or a sink, (as estimated by the cost model) it is designated as such.

4.4.3 Code motion

In order to keep the implementation of the code motion simple, the scheduler moves instructions one basic block at a time. The scheduler performs the required code motion using the set of upward and downward transformations illustrated in Figure 4-9 and Figure 4-10, respectively. Figure 4-9(a) shows an instruction, which modifies register r, being moved up from a basic block B2 to its parent basic block B1. B1 has another child, B3, and the move is valid only if r is dead at the top of B3. If B2 has more than one parent then the move is valid only if the condition is met for each parent. Figure 4-9(b) shows two copies of an instruction in basic blocks B2 and B3 being moved up and merged into one copy in parent basic block B1. If B1 has more children and each child has a copy of the instruction then all the copies can be merged into one copy in B1. If any child of B1 does not have a copy of the instruction, then the transformation in Figure 4-9(a) is applied. Figure 4-9(c) shows an instruction in basic block B3 being moved up and duplicated in parent basic blocks B1 and B2. If B1 or B2 have other children, then the transformation in

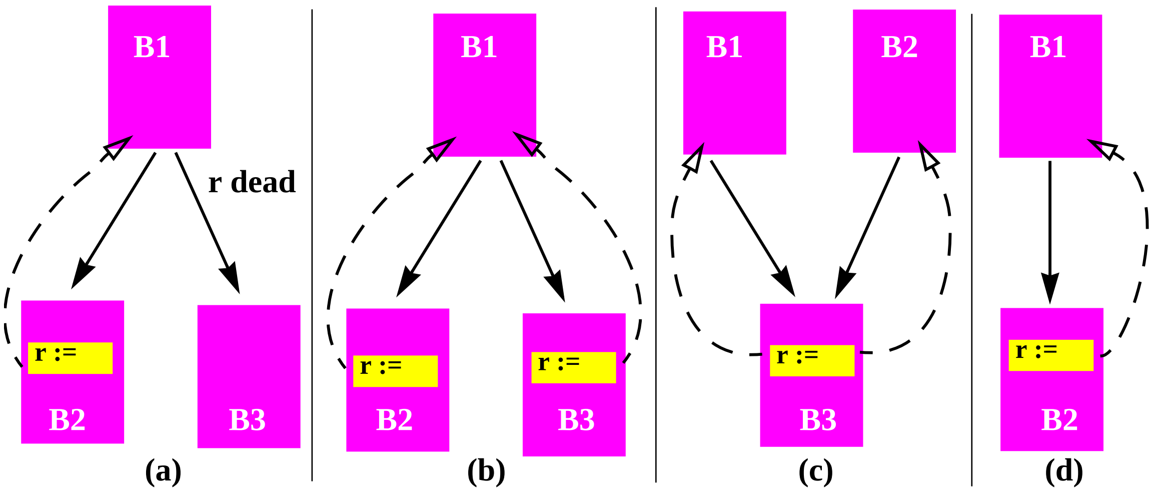


Figure 4-9: Upward transformations. ➔ indicates control flow. ⇾ indicates code motion. (a) An instruction is moved from a basic block to its parent if the destination register is dead down the other siblings of the basic block. (b) Two instructions that are copies are merged into a single copy in the parent basic block. (c) An instruction is duplicated into each of the parents. (d) An instruction is moved from a basic block to its only parent.

Figure 4-9(a) is applied. Figure 4-9(d) shows an instruction being moved up from a basic block, B2, to its only parent basic block, B1.

Figure 4-10(a) shows an instruction, which modifies register r, being moved down from basic block B1 to its child B3. B3 has another parent B2 and the move is valid only if r is dead at the top of B3. Figure 4-10(b) shows two copies of an instruction in basic blocks B1 and B2 being moved down and merged in child basic block, B3. If any parent of B3 does not have a copy of the instruction, then the transformation in Figure 4-10(a) is applied. Figure 4-10(c) shows an instruction being moved down from basic block B1 and duplicated in children basic blocks B2 and B3. If B2 or B3 have other parents, then the transformation in Figure 4-10(a) is applied. Figure 4-10(d) shows an instruction being moved down from basic block, B1, to its only child basic block, B2.

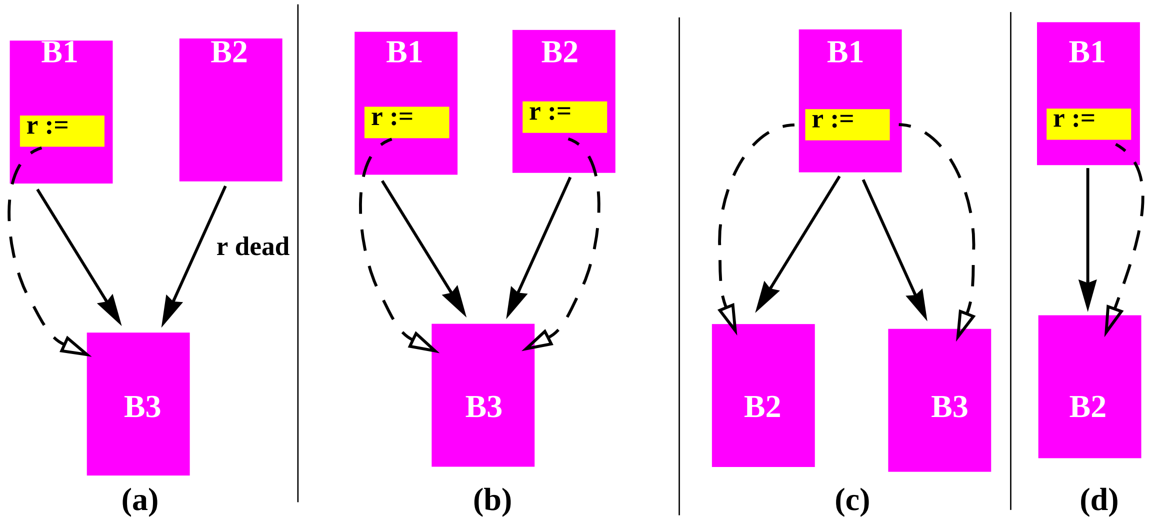


Figure 4-10: Downward transformations. → indicates control flow.⇾ indicates code motion. An instruction can be moved down from basic block A to basic block B only if the reaching definitions of the source operands of the instruction are the same in blocks A and B. (a) An instruction is moved from a basic block to its successor if the destination register is dead down the other sibling blocks. (b) Two instructions that are copies can be merged into a single instruction in the successor block. (c) An instruction is duplicated into each of the successor blocks. (d) An instruction is moved from a basic block to its only successor.

The actual code motion itself is performed in two phases. First, the sourcing instructions are moved up and then the sinking instructions are moved down. The scheduler moves the chosen instructions in the chosen direction one basic block at a time. In the upward phase, the scheduler starts with the leaf basic blocks of the task and works upward to the root of the task in reverse topological order. In every basic block, it pushes the sourcing instructions up and across the top of the basic block into the end of the predecessors of the current basic block. When all the sourcing instructions in the current basic block have been considered for upward motion, the next basic block in reverse topological order is visited. In the second phase, the scheduler starts with the root basic block of the task and works downward to the leaves of the task in depth-first-search order. In every basic block, it pushes the sinking instructions down and across the bottom of the basic block into the top of the successors of the current basic block. When all the sinking instructions in the current basic block have been considered for downward motion, the next basic block in depth-first-search order is visited.

4.4.3.1 Issues related to code motion

Static code motion is hindered by ambiguous memory dependences; loads and stores cannot be moved past stores unless it is guaranteed that the instructions do not access the same address. Although the ARB provides support for inter-task memory disambiguation, Multiscalar hardware does not support speculative code motion of ambiguous loads and stores within a task; if a load instruction is moved past a store instruction to the same address as the load, program semantics is violated. Better static memory disambiguation allows more loads and stores to be moved past other stores.

One of the important issues of static code motion is safety of transformations. Moving loads and stores past branches may cause spurious exceptions although the original program does not experience any exceptions. For example, if a load is moved up past a branch, then the load may dereference a null pointer, while in unmodified code, execution branches around the load. The two key issues are: (1) differentiating between spurious (caused by “unsafe” global code motion) and real exceptions, and (2) recovery of the correct state on a real exception. The Multiscalar architecture provides checkpointed state at the beginning of every task. When an outstanding exception (which are assumed to be infrequent) is signalled, execution may be rolled back to the start of the task and a copy of the original code (without any scheduling transformations) may be re-executed. During the rerun of the original code, if the exception occurs again, then it may be taken because re-execution of the original code guarantees that only real exceptions occur. The address of the task corresponding to the original code may be included in the task descriptor and the code itself may be attached at the end of all scheduled tasks.

In the context of scheduling for superscalar processors, many techniques have been proposed to solve this problem. An approach similar to those proposed in boosting [93] and sentinel scheduling [66] may be adopted. Instructions that have not been moved may be annotated to signal any outstanding real exceptions. Bookkeeping code may be generated to restore precise state to handle real exceptions.

4.4.4 Cost model

The scheduler employs a cost model to determine the extent to which instructions have to be moved. The cost model uses static or dynamic profiling information to determine the branch taken and branch not-taken probabilities. The cost model estimates the stall cycles incurred by each forwarding (receiving) instruction by estimating the difference between the time of execution of the forwarding (receiving) instruction and the time of execution of the sinking (sourcing) instruction that consumes (produces) the register value produced (consumed) by the forwarding (receiving) instruction. The time of execution of an instruction contained in a task is estimated as the number of dynamic instructions executed before the instruction from the start of the task and is called the **position** of the instruction.

To the first order, the number of dynamic instructions executed before an instruction is a reasonable estimate for the time of execution of the instruction with respect to the start of its task. Instructions that take multiple cycles (e.g., loads) to complete introduce error in this estimate, but obtaining accurate profiling information about cycle counts for such instructions is difficult because the profiler may have to perform a detailed timing simulation of the target hardware. Using dynamic instruction count is convenient because the target system need not be simulated.

The estimation is based on the equations given in Figure 4-11. *Pathprob* of a basic block i is the probability that the basic block i is executed if the task to which the basic block belongs is executed, without including back edges of the CFG of the task. *acycliclen* of a

$$\text{pathprob}(i) = \sum_{j \in \text{parent}(i)} \{\text{pathprob}(j) \times \text{prob}(i, j)\}$$

$$\text{acycliclen}(i) = \sum_{j \in \text{parent}(i)} \{\text{acycliclen}(j) \times \text{pathrob}(j)\}$$

$$\text{looplen}(l) = \text{acycliclen}(\text{loopend}(l)) / (1 - \text{backprob}(l))$$

For the nth static instruction in the task which is not at the top of its basic block, i:

$$\text{position}(I_n) = \text{position}(I_{n-1}) + \text{pathprob}(i), \text{postion}(I_0) = 0$$

For the nth static instruction in the task which is at the top of its basic block, i:

$$\text{position}(I_n) = \sum_{j \in \text{parent}(i)} \{\text{position}(I_{\text{end}(j)}) \times \text{pathprob}(j)\} + L(i)$$

For a basic block i that belongs to a possibly nested loop, where l is the outermost loop:

$$L(i) = \text{looplen}(l)$$

Figure 4-11: Cost model. *prob(i,j)* is the probability that control flow edge (i,j) is taken during execution; it is obtained through profiling. *pathprob* of basic block *i* is the probability of reaching *i* from the root of its task. *acycliclen* of loop *i* is the average number of instructions executed by a single iteration of *i*. *looplen* of loop *l* is the average number of instructions executed by the loop. position of an instruction is the average number of instructions executed by the task before the instruction. *end(j)* is the static location of the last instruction in basic block *j*. *L(i)* is 0 if *i* does not belong to any nested loops within the task.

basic block i is the average dynamic number of instructions executed before basic block i is reached in the task, without including back edges of the CFG of the task. *Prob(i,j)* is the probability of the edge (i,j) in the CFG being taken during execution. If a task does not include any loops, then position of the instructions in the task is estimated by *acycliclen*. In order to account for loops in tasks, a hierarchical approach is taken. Loops are considered from inner most nesting depth to outer nesting depth. The average number of dynamic instructions executed by a loop is estimated by using the expression for *looplen*.

Going from inner to outer loop nesting in the task, loops are replaced with their equivalent *looplen* and the *acycliclen* of the next outer loop nest is computed. The process terminates when *acycliclen* of the outermost level is computed. Finally, computing the position of an instruction depends on its location within its basic block. If an instruction is not at the top of its basic block, then its position is the sum of the position of the previous instruction in the basic block and pathprob-weighted count of one instruction. If an instruction is at the top of its basic block then its position is the pathprob-weighted average of the last instruction of predecessor basic blocks. All possibly loops included within tasks are replaced by the equivalent *looplen* of the outermost loop. For all instructions in a possibly nested loop, *looplen* of the outermost loop is added to compute their position.

After the position of the instructions is determined, the scheduler moves the forwarding (and the sourcing) instructions up and the receiving (and the sinking) instructions down until the position of the forwarding instruction is smaller than that of the receiving instruction or the instructions cannot be moved any further. When the instructions are moved via the upward and downward transformations, their position is swapped with the position of the instruction they move past, so that code motion is stopped as soon as a good schedule is achieved. Figure 4-6 shows an example of register communication scheduling. Task 1 computes and forwards register r and Task 2 uses the value. The scheduler moves the forwarding instruction ($r := a + b$) up into the predecessor basic block by applying the transformation of Figure 4-9(a). The forwarding instruction and a sourcing instruction ($a :=$) are moved up Task 1 by applying the transformation of Figure 4-9(d). Let us assume that the instructions in Task 1 cannot be moved any more. In Task 2, the scheduler moves the receiving instruction ($c := r + s$) down into the successor basic block by applying the transformation of Figure 4-10(d). The receiving instruction and a sinking instruction ($x := c + d$) are moved down into the successor basic blocks by applying the transformation of Figure 4-10(c).

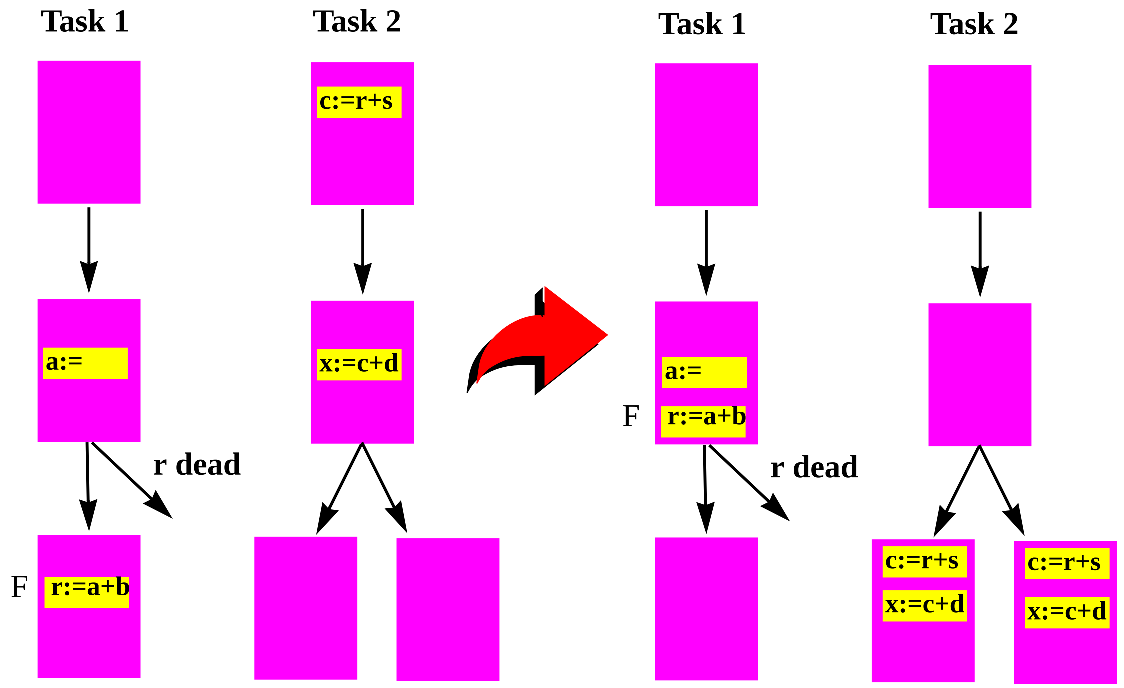


Figure 4-12: An example of register communication scheduling. Register r is produced (and forwarded) by Task 1 and consumed by Task 2. The forwarding instruction (r := a + b) along with one other sourcing instruction (a:=) is moved up Task 1. The receiving instruction (c := r + s) and a sinking instruction (x := c + d) is moved down to the bottom of Task 2.

The scheduling requirements of multiple instructions may conflict with one another. It is possible that minimizing the stall due to one dependence may actually increase the stall caused by another dependence. Similar to task selection, the scheduler also prioritizes dependences and the corresponding instructions, based on statically or dynamically profiled execution frequencies and performs scheduling in the order of priority. Instructions with lower priority are scheduled before instruction with higher priority, so that the schedule of the more important instructions are not disrupted by the others.

4.4.5 Controlling instruction count increase

After an instruction is identified to be moved, it is moved in the appropriate direction as long as the estimated stall is non-zero. The cost model tracks the change in the estimated stall cycles of the instructions through all the transformations. Thus, the cost model determines which direction an instruction needs to be moved and how far it needs to be moved. Moving an instruction past the point of zero stall cycles is undesirable because any further code motion may increase the count of dynamic instructions without any improvement in the number of stall cycles (transformations of Figure 4-9(a) and Figure 4-10(a) may increase the instruction count because they may move an instruction into paths that did not execute the instruction before).

There are many ways to control this increase. One of the guiding principles in controlling the instruction count increase is that the only goal of upward motion is to hoist forwarding instructions up. If for some reason, the forwarding instruction does not move past a basic block, then it is detrimental to move the sourcing instructions in its backward slice past that basic block because that may increase the instruction count without hastening the forwarding instruction. A similar argument applies to downward code motion as well; if for some reason, a receiving instruction does not move past a basic block, then it is detrimental to move the sinking instructions in its forward slice past that basic block. Before applying the transformations, a look-ahead is performed within the current basic block to check if for every sourcing(sinking) instruction, at least one of the forwarding(receiving) instructions in its backward(forward) slice will be eventually be able to move up(down) across the top(bottom) of current basic block. If not, then the sourcing(sinking) instruction is not moved up. The look-ahead needs to be done only within the current basic block.

There is an important interaction between control flow and register communication which may also be exploited to control instruction count. If there is a define of a register down one of the paths past a branch and none down the other path, then the register cannot be forwarded/released before the branch because only after the branch, there is a guarantee that the register is not modified down the other path. This observation implies that when hoisting a forwarding instruction up past a branch from one path, if the other path past the branch defines the register then this hoisting will be detrimental. If there is a use of a register in all the paths reaching a receiving instruction (from the root of its task) that consumes the register, then there is no advantage in pushing the receiving instruction down. Figure 4-13 illustrates these two arguments.

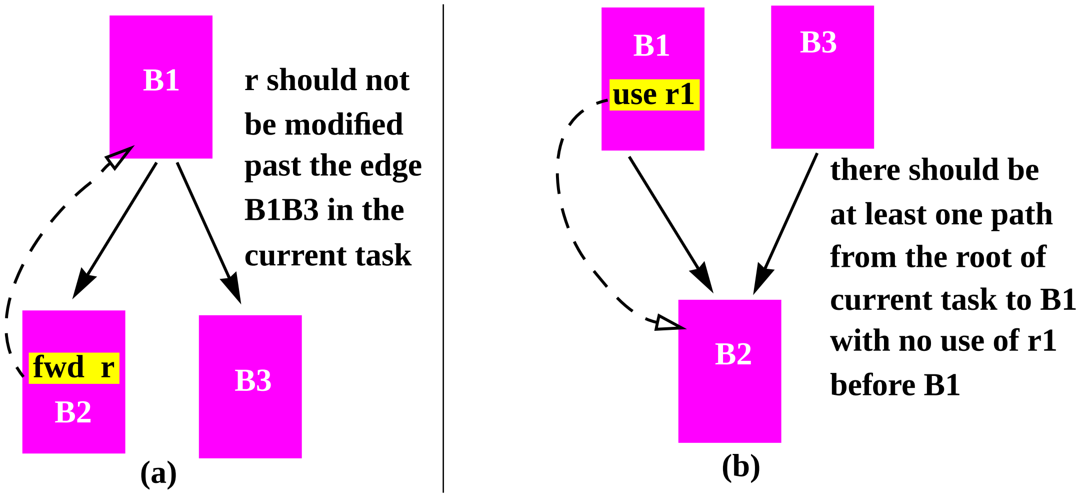

B1
r should not be modified past the edge B1B3 in the current task
fwd r
B2
B3
(a)
B1
use r1
B3
there should be at least one path from the root of current task to B1 with no use of r1 before B1
B2
(b)

Figure 4-13: Conditions to prevent unnecessary dynamic instruction increase. ➤ indicates control flow. –▷ indicates code motion. (a) If a forwarding instruction is moved up from block B2 to block B1, then there should be no other definitions of r past B1 in the current task. (b) If an instruction consuming r1 is moved from block B1 to block B2, then there should be no previous use of the r1 in all paths reaching B1 from the root of the current task.

Apart from these considerations, the cost model is also used to effectively control the instruction count increase. Associated with all the relevant instructions is the estimated number of stall cycles that the instruction would cause and throughout the code motion, this number is tracked for each instruction that is moved and as soon as the estimated number of stall cycles has decreased to a zero, code motion terminates.

4.4.6 Scheduling loop induction variables

Induction variables of loops are an important special case for scheduling register communication. Specific control flow pattern of loops can be exploited to schedule induction variables more effectively through a different technique. If the entire loop body is selected as a task, then the different iterations of the loop are assigned to different PUs to be executed in parallel. But induction variables are usually produced at the end in one iteration and used in the next iteration right at the beginning. This code arrangement causes each iteration to wait until the previous iteration sends the induction registers, serializing the iterations. Since the sequence of values that induction variables take is predictable at compile time, a better schedule can be obtained. Induction variables can be forwarded early by restructuring the loop so that the induction variables are incremented first and then used by the loop body; the loop is restructured so that the induction variables are incremented at the top of the loop. The exit-test branch also gets moved to the top of the loop and an extra jump is added at the bottom of the loop. The first iteration is peeled off without the induction variable increments, to maintain correctness. The restructured loop enables each iteration to forward the induction registers in the first few instructions, supplying the next iteration with the values it needs without any delay. Figure 4-14 shows the general scheme of restructuring loops.

4.4.7 Interaction with other optimizations

Register communication scheduling interacts with live ranges of registers and pipeline delay-slot scheduling. Apart from these interactions, register communication scheduling has phase ordering problems with register allocation and task selection.

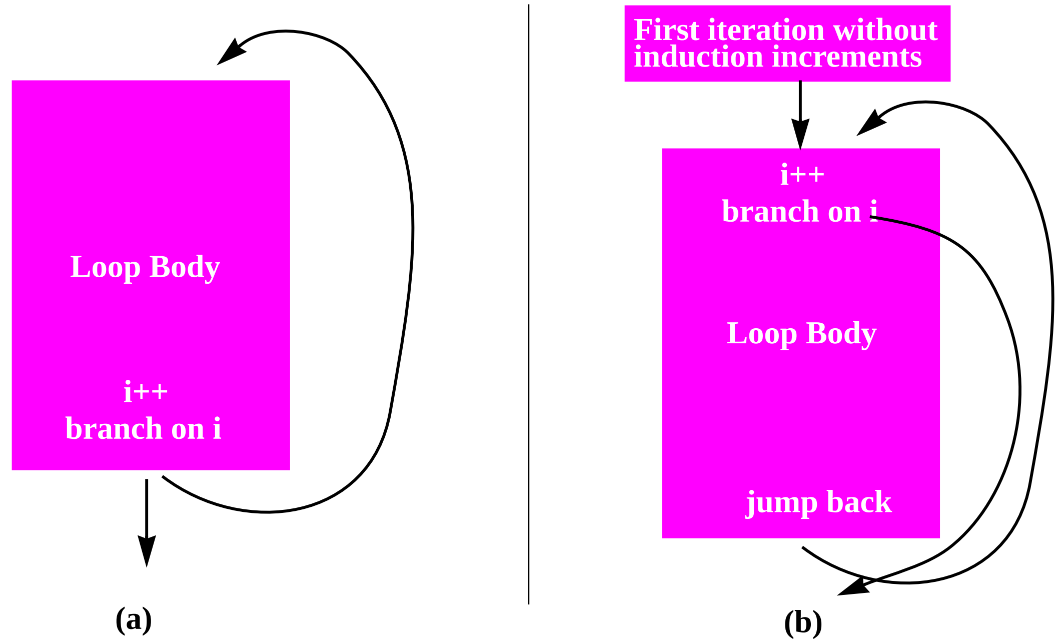


Figure 4-14: Induction variables in loops. (a) Increments of induction variables at the end of the loop body serializes the loop iterations. (b) Loops are restructured so that induction variable increments and the branch exit test are moved to the top of the loop and an extra jump is added at the bottom. To maintain the same number of iterations, the first iteration is peeled off.

Any instruction scheduling, in general, fundamentally changes the expanse of live ranges of registers. Any expansion of live ranges increases register pressure and may result in increased spilling. The set of transformations employed by register communication scheduling expand the live ranges of one set of registers and contract the live ranges of another set of registers. Defines and uses of the registers that are communicated from one task to another are moved up and down control flow, respectively, leading to expansion of the live ranges of the registers. Defines and uses of the registers in dependence chains that start with a receiving instruction and end with a forwarding instruction are moved toward each other, leading to contraction of the live ranges of the registers.

Register communication scheduling may disrupt the pipeline/superscalar schedule of the code, resulting in pipeline bubbles. In the current implementation, integrating pipeline/superscalar scheduling with register communication scheduling may address the issue.

I implemented the scheduling algorithm before the register allocation phase. The advantage of such phase ordering is that there are very few anti-dependencies between instructions with pseudo-register operands and renaming does not have to consider the problem of spilling registers. The disadvantage of such phase ordering is that anti-dependencies introduced by register assignment may delay forward of a register, (as described in Section 4.3.2) which is taken into account by the scheduler. Round-robin assignment of registers may mitigate the problem. As with many other compiler optimizations, better memory disambiguation would probably result in better scheduling.

There is a phase ordering problem between task selection and register communication scheduling; tasks should be selected so that a good schedule of any inter-task register communication is obtained but inter-task register communication can be identified only after tasks are selected. In the current implementation, task selection is done before register communication scheduling is done. Task selection attempts to avoid many inter-task data dependences by including as many data dependences as possible within tasks. After task selection is done, those data dependences that are not successfully included within tasks are handled by the scheduler to mitigate the detrimental effects of inter-task register communication. Other phase-ordering or an integrated task selection and scheduler may be possible.

4.5 Implementation

The most important part of generating register communication is to solve the dataflow equations in Figure 4-3 and Figure 4-4. The equations compute the dataflow properties of basic blocks and not individual instructions. Individual instructions are annotated to orchestrate register communication. The dataflow property *Send* identifies the basic blocks which should send register values of the registers they define. Within each such basic block, the instructions are scanned and the instruction that defines the appropriate register last is annotated to send the register value. Figure 4-15 shows the pseudo-code to generate register communication. *Annotate_reg_comm()* scans through every task and annotates the appropriate instructions. *Solve_equations()* computes the dataflow property *Nomoredef*, as per the equations in Figure 4-3. *Solve_equations()* solves for *Nomoredef* by iterating over

```
annotate_reg_comm() {
  for each task t {
   solve_equations();
   for each basic block b{
         if (send(b,r) &&
                     not_write(b,r))
               insert_release(b,r);
         for each instruction i in b {
               if (send(b, r) &&
                     last_write(i,r))
                annotate_forward(i);
         }
   }
  }
}
```

```
solve_equations() {
   change = TRUE;
   while (change) {
    change = FALSE;
    for each basic block i, last to first {
         for each child j of i {
   if  $Nomoredef(i) \not\subset Nomoredef(j) \cap BBdef(j)$
   {
               change = TRUE;
             $tmp = Nomoredef(j) \cap BBdef(j)$
         $Nomoredef(i) = Nomoredef(i) \cap tmp$
   }
    }
   }
   for each basic block i, first to last {
    for each parent j of i {
      $tmp = Nomoredef(j) \cup tmp$
    }
     $send(i) = Nomoredef(i) \cap tmp$
   }
}
```

Figure 4-15: Pseudo-code for register communication generation. *Annotate_reg_comm*() is the top level driver for register communication generation. *solve_equations*() solves the dataflow equations given in Figure 4-3.

Nomoredef until convergence. Since *Nomoredef* is a backward flow problem, *solve_equations()* iterates over basic blocks in reverse CFG depth-first-search numbering order to converge quickly.

Figure 4-16 shows the pseudo-code for scheduling register communication. The sinking and the sourcing instructions that need to be considered for scheduling are identified by simple dataflow dependence analyses. *Sort_dependences()* prioritizes the dependences. Using the expressions in Figure 4-11, *compute_position()* determines the position of each instruction in the task. Then the instructions are scheduled in increasing order of priority by using the transformations in Figure 4-9 and Figure 4-10. *move_code()* moves the sourcing instructions up and the sinking instructions down until the position of the forwarding instruction is smaller than that of the receiving instruction. *move_up()* moves instructions up from one basic block to its predecessors. *move_up()* performs the transformations shown in Figure 4-9 and Figure 4-10. *move_up()* moves up all instructions in the basic block that the forwarding instruction is dependent on and then moves the forwarding instruction up. *move_up()* moves each of the sourcing instructions up into predecessor basic blocks only if all of the sourcing instructions in the basic block can be moved up into predecessor basic blocks; even if one of the sourcing instructions cannot be moved up, then none of them is moved. *move_down()* is similar to *move_up()*.

4.6 Related work

Dataflow analyses have been applied to a large number of problems in compiler optimizations. The basic register communication problem falls under the category of simple dataflow problems [1] [28] [30] [72]. Compilers of data parallel languages like Fortran D and HPF generate data communication for distributed memory machines [49] [51]. Some of the correctness criteria of Multiscalar register communication and distributed memory communication are similar. The important differences are that multiple sends of a value is disallowed in distributed memory communication. And these works do not include any speculative communication.

```
schedule_comm() {
        sort_dependences();
        compute_position();
        for each dependence (u,v) {
         move_code(u,v);
        }
}

compute_position() {
     for each task t {
         for each level of nesting L {
             compute_acycliclen();
         }
         for each instruction i in task t {
             consume(i) = acycliclen(i);
             produce(i) = looplen(i);
         }
     }
}

move_code(p, c) {
  while (consume(c) < produce(p)) {
      move_up(p);
      move_down(c);
      if (cannot_move(c) &&
                  cannot_move(p))
             break;
  }
}
```

```
move_up(i) {
   for each instruction j in basic block {
      if (dependent(i, j)) {
           move_to_top_of_basicblock(j);
           for each parent basicblock p of i {
            if !(legal(j, p) && instr_count_ok(j, p))
                return FAIL;
           }
       }
   }
    for each parent basicblock p of i {
            if (dependent(i, j))
                make_copies(j, p);
    }
 }

move_down(i) {
   for each instruction j in basic block {
       if (dependent(i, j)) {
           move_to_bottom_of_basicblock(j);
           for each child basicblock of i {
            if !(legal(j, c) && instr_count_ok(j, c))
                return FAIL;
           }
       }
      }
    for each child basicblock c of i {
      if (dependent(i, j))
           make_copies(j, c);
    }
 }
```

Figure 4-16: Pseudo-code for scheduling register communication. *Schedule_comm*() is the top level driver for the scheduler. *compute_position*() computes the position of the various instructions. *move_code*() moves the instructions in the appropriate direction. *move_up*() and move_down() perform the actual transformations.

Annotating instructions with register communication information has precursors. Other approaches like boosting [92] [93] have annotated instructions with extra information.

There are a myriad of compiler scheduling techniques for superscalar, VLIW and pipelined architectures. In general, most compiler scheduling techniques have two components: (i) the set of transformations used for code motion, and (ii) the cost model used to decide when to apply the transformations. The purpose of most scheduling techniques like instruction scheduling [17] [41] [43], global scheduling [15] [71], percolation scheduling [76], trace scheduling [34], balanced scheduling [61], boosting [92], hyperblock scheduling [67], and superblock scheduling [22] is to fill a given empty issue slot with a profitable instruction. The set of transformations create a large set of ready instructions, and the cost model decides the most profitable instruction among the ready instructions to fill the empty slot. In contrast, register communication scheduling requirement is to move a given instruction to a suitable position. The set of transformations used by register communication scheduling can move the instruction to a number of different positions. Register communication cost model decides the most profitable position among the possible positions to move the instruction. Thus, there is a reversal of what is fixed (or given) and what is varied. Moreover, conventional scheduling techniques do not move entire slices of computation. The cost models of conventional techniques compute the probability of execution of the ready instructions with respect to the empty slot and choose the most profitable ready instruction. Register communication cost model estimates the delay incurred by forwarding/receiving instructions and decides which direction the instruction has to be moved. Register communication cost model also controls the extent to which each instruction gets moved.

Static speculation has been applied in scheduling for superscalar, VLIW and superpipelined machines. Techniques like global scheduling [15], percolation scheduling [76], trace scheduling [34], boosting [15] [92], hyperblock scheduling [16], superblock scheduling [17], guarded execution [82], sentinel scheduling [66], and modulo scheduling [83] employ varying degrees of static speculation. Utilizing speculative execution in instruction scheduling constitutes two issues: (i) decision of when to perform computation under speculation, and (ii) recovery from incorrect speculation. The first issue applies to compiling speculative register communication in the Multiscalar architecture as well. The second issue is addressed by a combination of hardware and compiler. The compiler inserts code (in the form of “re-sends”) to detect any incorrect static speculation and the hardware performs recovery. Unlike most VLIW machines, no bookkeeping code need be inserted in the program but recovery may be more expensive since larger number of instructions may get squashed.

Loop restructuring is similar to some of the transformations done by software pipelining techniques [32] [64] [83] to prepare the unrolled loop for software pipelining.

Work in parallelization and scheduling of Do-across loops [23] [26] [27] generate synchronizations between iterations of loops that allow arbitrary communication between iterations. The major difference between our problem and doacross parallelization/scheduling is that Multiscalar tasks are arbitrary sub-graphs of the control flow graph and need not be loop bodies. Moreover, the machine models of the doacross work do not include control and data speculation, which are the default in the Multiscalar architecture.

Although, I have described our scheduling technique in the context of the Multiscalar architecture, it can be applied to other architectures like dynamic trace-based processors [86] [104], multithreaded architectures [103], small scale fine-grain parallel architectures [78] and any architecture with multiple flows of control and fine-grain data communication. In the Multiscalar architecture, the unit of work executed on each PU is a task, whereas in multithreaded architectures, it is a thread. The threads in multithreaded architectures communicate values that may be fairly small in grain size and performance requirements of streamlining the communication are similar to those in the Multiscalar architecture. In multithreaded architectures also, producers have to be scheduled early and consumers have to be scheduled late. The cost model to estimate the delays caused by communication fits in the multithreaded paradigm as well. Register communication scheduling technique may be applied to these other architectures.

Chapter 5

Experimental Evaluation

An important goal of this research is to construct a compiler for the Multiscalar architecture so that various hardware and compiler techniques are experimented with and evaluated on large benchmarks. To that end, the optimizations described in the preceding chapters have been implemented in the Gnu C Compiler, gcc. I conducted many experiments to determine the effectiveness of the compiler techniques by performing extensive simulations on a simulator that faithfully represents the Multiscalar hardware. In Section 5.1, I motivate and describe the measurements I make. In Section 5.2, I describe the experimental framework and list the parameters of the hardware and the compiler configurations used in the experiments. In Section 5.3, I describe the experiments measuring various aspects of task selection and register data communication and present results of the effect of each of the compiler optimizations on overall performance. In addition to performance, I present measurements of selected metrics that isolate the impact of each optimization. I study the interaction between relevant hardware optimizations and compiler optimizations. I also present analyses of the results to correlate program behavior, effectiveness of compiler optimizations, and overall performance.

5.1 Overview of experiments

In this section, I describe the quantities measured in the experiments with the benchmarks from the SPEC95 [98] suite to analyze performance issues of the Multiscalar architecture and demonstrate the impact of the compiler techniques. I evaluate: (1) overall performance, (2) task characteristics relevant to performance, and (3) dynamic window established.

In order to understand the relative importance of the various task selection heuristics, I isolate the performance achieved by each heuristic by presenting successive improvements in performance. I generate tasks using the heuristics described in Chapter 3. I start with tasks made of single basic blocks and progressively add control flow heuristic and then data dependence heuristic. Basic block tasks are simple and serve as a natural yardstick to experiment with extraction of parallelism. Recall that control flow heuristic generates tasks with at most some fixed number of targets taking advantage of reconvergent control flow paths and loops and data dependence heuristic generates tasks by including data dependence chains[1] entirely within. Task size heuristic includes short function calls within tasks to avoid terminating tasks at all function calls. Setting the threshold level to include entire function invocations, which execute fewer than 15 instructions, within tasks avoids generating tasks that incur frequent ARB overflow, inordinate number of memory dependence squashes, and excessive load imbalance.

Then, I analyze the performance achieved by each benchmark in light of its task characteristics and its average window span. The important task characteristics that affect performance of the Multiscalar architecture are: task size, inter-task control flow, and inter-task data dependence.

[1] The current implementation handles register dependence chains and simple memory dependences.

For each task characteristic, I measure a metric that closely correlates to performance. I measure the average dynamic task size. For inter-task control flow, I measure prediction accuracies to estimate the magnitude of control flow misspeculation. I also present the distribution of the number of targets actually taken by the benchmark to determine the kind of targets that the heuristics expose to the prediction hardware. For inter-task data dependence, I report the time spent by an instruction waiting for register values from other tasks. I also measure the number of dynamic inter-task data dependencies (register and memory), i.e., the number of instances where a value produced by one task is consumed by another.

Due to loop restructuring, which involves inverting loop bodies and adding a jump instruction at the bottom of the loop, and register communication generation, which involves adding release instructions in some control flow paths, Multiscalar programs may execute more instructions than the superscalar versions. I present the dynamic instruction count overhead for the Multiscalar versions.

By studying the nature of the dynamic window established by Multiscalar organizations, we can estimate the amount of parallelism that the machine can exploit. For superscalar machines, the average window size is a good metric of quality of the dynamic window. Since the Multiscalar architecture does not establish a single continuous window, I extend window size to another metric. For the Multiscalar architecture, the total number of dynamic instructions that belong to all the tasks in execution simultaneously called the **window span** is a metric that indicates the amount of expected overlap among the execution of the instructions. Window span captures the amount of parallelism exploited by the machine. I present the average window span for the benchmarks and correlate it to performance.

Register communication delay and register communication traffic demand significantly affect overall performance of the Multiscalar architecture. I have implemented various compiler techniques to reduce the delay and the traffic. Register communication strategies and scheduling discussed in Chapter 4 reduce the delay. The various register communication strategies offer a spectrum of hardware and compiler involvement with varying degrees of performance. Recall that end_send strategy conservatively posts forwards of register values at task end, eager_send strategy posts speculative forwards of register values as soon as a register is modified, even though it may not be the last modification, last_send strategy posts forwards of register values only after the last modification, and spec_send strategy posts speculative forwards of register values after a register is modified, if it is likely to be the last modification. I quantify the effect of register communication delay by studying impact of the various register communication strategies on overall performance. To isolate the impact of the various strategies, I also present measurements of the number of cycles spent by PUs waiting for a register value to be sent.

Register communication scheduling further reduces the delay by overlapping communication with computation. In addition to the communication strategies, I also measure the impact of register communication scheduling on performance. To isolate the impact of scheduling, I measure of the number of cycles spent by PUs waiting for a register value to arrive.

Register communication bandwidth is an important resource that needs to be utilized judiciously. High bandwidth demand could lead to queuing delay in register communication leading to worse performance. To this end, I present measurements of register communication traffic of the benchmarks and the impact of dead-register optimization on the traffic.

5.2 Experimental methodology

Before presenting experimental results, I describe the experimental framework, the parameters values used to configure the hardware simulator and the benchmarks used in the evaluations.

5.2.1 Framework

I implemented a C compiler derived from the Gnu C Compiler, *gcc*. Benchmark source files are input to the compiler which produces executables. To evaluate compiler strategies described before, I used an instruction driven simulator of the Multiscalar architecture. The binary generated by the compiler is executed by the simulator which faithfully captures the behavior of the Multiscalar architecture on a cycle per cycle basis and produces output result files for verification. The simulator executes all instructions of the benchmarks except for system calls, in order to maintain high accuracy of performance results obtained.

5.2.2 Simulation parameters

The simulator models details of the processing units, the sequencer, the control flow prediction hardware, the register communication ring, the memory hierarchy consisting of the ARB, L1 data and instruction cache, L2 cache and main memory. Apart from these hardware components the simulator models the interconnection between the processing units and the ARB and L1 instruction and data caches, the L1 caches and the L2 caches, as well as the system bus connecting the L2 caches and the main memory. Both access latencies and bandwidths are modeled at each of these components and interconnects.

5.2.2.1 Processing units

The PUs issue instructions out-of-order or in-order using a register update unit with 16 reorder buffer entries and an issue list of 8 instructions. Each PU can fetch up to 4 instructions, issue up to 2 instructions, and commit up to 4 instructions, in one cycle, respectively. Each PU executes instructions on a collection of fully pipelined functional units: 2 simple integer FUs, 1 complex integer FU, 1 floating point FU, 1 branch FU, and 1 memory FU. For intra-task branch prediction, the PUs use a *gshare* branch prediction scheme [70] with a 16-bit history register indexing into a 64K-entry table of 2-bit counters. Since the main objective of the Multiscalar architecture research is to design distributed PUs each of which may be clocked fast, I use narrow-issue PUs for my experiments.

5.2.2.2 Control flow prediction

The global sequencer employs a 2-way set associative, 32KB task cache with 64 byte blocks for task descriptors. On a hit, the task cache returns a task descriptor in one cycle. So, tasks are initiated at most one per cycle. Misses from the task cache are sent to the common L2 cache. To assign tasks to PUs, the control flow prediction hardware of the sequencer uses a dynamic path-based [56] scheme to predict one out of at most four targets of tasks. The path-based predictor uses 7 path histories XOR-folded into a 16-bit of path register indexing into a 64K-entry table of 2-bit counters and 2-bit target numbers. The path based-predictor is complemented by a 64 entry return-address stack to handle function call returns.

5.2.2.3 Register communication

Each PU can send as many register values as the number of instructions that can be issued in one cycle. Outputs of functional units of one PU available at the end of a cycle are bypassed to the inputs of functional units of the next PU in the beginning of the next cycle through the unidirectional point-to-point register communication ring.

5.2.2.4 Memory subsystem

The PUs fetch instructions from a 2-way associative, fully pipelined, lockup-free [62] [96], shared L1 instruction cache with 32 byte blocks interleaved into as many banks as there are PUs. Each cache bank contains 16KB of storage. For 4-PU and 8-PU configurations, the size of the L1 I-cache is 64KB and 128KB, respectively. The L1 I-cache banks are connected to the PUs through a crossbar and contention at the banks is modeled. On hits, the L1 I-cache returns 4 instructions to the requesting PU in one cycle, if there is no contention at the banks. Although crossbar traversal and I-cache access in one cycle may seem optimistic, another organization of private I-caches with each PU may make the assumption feasible.

The PUs access a 2-way associative, fully pipelined, lockup-free, shared L1 data cache with 32 byte blocks interleaved into as many banks as there are PUs. Each cache bank contains 16k of storage. For 4-PU and 8-PU configurations, the size of the L1 D-cache is 64KB and 128KB, respectively. The L1 D-cache banks are connected to the PUs through a crossbar and contention at the banks is modeled. On hits, the L1 D-cache returns a word of data to the requesting PU in two cycles, if there is no contention at the banks. I assume a 2-cycle hit latency to account for the crossbar traversal and the D-cache access.

Loads and stores from the PUs access the ARB and the L1 D-cache simultaneously. The ARB is a fully associative, fully pipelined, lockup-free, shared buffer interleaved into as many banks are there are PUs. Each ARB bank contains 32 entries each of which have as many blocks are there are PUs and the ARB blocks are of the same size as L1 D-cache blocks. For 4-PU and 8-PU configurations, the size of the ARB is 4KB and 8KB, respectively. On hits, the ARB returns a word of data to the requesting PU in 2 cycles. The ARB banks are connected to the PUs through a crossbar and contention at the banks is modeled. On task squashes, the ARB entries are invalidated in one cycle and on task commits, dirty ARB entries are written back to the L1 D-cache in the background[2], overlapped with the execution of the next task assigned to the PU.

Misses from the L1 I-cache, L1 D-cache and the ARB are sent to the unified L2 cache. The L2 cache is a 2-way associative, fully pipelined, lockup-free cache interleaved into 4 banks with 128 byte blocks. Each bank contains 1Mb of storage. For both 4-PU and 8-PU configurations, the size of the L2 cache is 4MB. The L2 cache hits return data to the requesting PU after 12 cycles, if there is no contention. The transfer rate between the L1 caches and the L2 cache is 16 bytes per cycle. The L1 caches and the L2 cache are con-

[2] Task commits do not incur the latency of writing back dirty ARB entries.

nected through a bus (assumed to be running at the same clock rate as the PU) and contention at the bus and the L2 cache banks are modeled. I assume a 12 cycle hit latency to account for L1 miss, L1-L2 bus protocol, L2 cache access, L1-L2 bus transfer and data return to PU.

Main memory is 4-way interleaved and is connected to the L2 cache through the system bus. Main memory is assumed to be infinite sized. Main memory returns data to the requesting PU after 58 cycles, if there is no contention. The transfer rate between the L2 cache and the main memory is 16 bytes per system bus cycle. The system bus is assumed to be clocked at half the clock rate of the PU. I assume a 58 cycle access latency to account for the L1 miss, L1-L2 bus acquire, L2 miss, system bus protocol, main memory access, system bus transfer, L1-L2 bus transfer and data return to PU.

Apart from the usual components of the memory hierarchy, the implementation of the Multiscalar architecture that I consider includes a hardware mechanism to perform memory dependence prediction and synchronization [73]. Instead of blindly speculating that loads and stores from different tasks are always independent (and squashing on misspeculations), the mechanism predicts dependences among memory accesses (and resulting misspeculations) by dynamically tracking the history of previously misspeculated memory dependences, similar to branch prediction. The mechanism enforces producer-consumer ordering on those loads and stores that are predicted to misspeculate by dynamically synchronizing them. The hardware tables used to perform memory dependence prediction contains 256 entries.

Hardware parameters are summarized inTable 5-1.

5.2.2.5 Compiler optimizations

All of the binaries for the experiments are generated with the highest level of gcc 2.7.2 optimizations. The usual compiler phases of jump optimization, common sub-expression

Table 5-1: Hardware parameters used in experimental evaluation.

Component	Description
PUs	2-way issue, 16-entry reorder buffer, 8-entry ready list
FUs	2 integer, 1 complex, 1 floating point, 1 branch, 1 memory
Intra-task prediction	gshare with 16-bit history, 64K-entry table of 2-bit counters
Inter-task prediction	path-based with 16-bit history, 64K-entry table of 2-bit counters and 2-bit target numbers
Task cache	32KB, 2-way associative
Register Ring	2 values per cycle, bypass same cycle between adjacent PUs
L1 I-cache	64KB (4PU)/128KB (8PU), 2-way associative, 32 byte blocks, 1 cycle hit, interleaved as many banks as the number of PUs, lock-up free, fully pipelined
L1 D-cache	64KB (4PU)/128KB (8PU), 2-way associative, 32 byte blocks, 2 cycle hit, interleaved as many banks as the number of PUs, lock-up free, fully pipelined
ARB	32 entries/PU, 32 x #PU bytes/entry, 4KB (4PU)/8KB (8PU), fully associative, 2 cycle hit, interleaved as many banks as the number of PUs, lock-up free, fully pipelined
Memory synchronization	256 entries
L2 cache	4MB, 2-way associative, 12 cycle hit, 16 bytes per cycle transfer
Main memory	Infinite, 58 cycle latency, 8 bytes per cycle transfer

elimination, dead code elimination, loop optimizations including strength reduction, induction variable elimination and loop unrolling, basic block pipeline scheduling, global register allocation, and peephole optimizations are used. Multiscalar optimizations including task selection, loop restructuring, dead register analysis for register communication, and register communication scheduling and generation are also used. The compiler uses basic block frequency, obtained via dynamic profiling, for register communication scheduling and task selection. Profiling was done using profile inputs specified by the SPEC95 suite.

5.2.3 Benchmarks

I use the SPEC95 benchmark suite throughout my experimental evaluation. The SPEC95 suite consists of both integer and floating point programs, drawn from a variety of application areas. The integer programs (CINT95) are 099.go, 124.m88ksim, 126.gcc, 129.compress, 130.li, 132.ijpeg, 134.perl, and 147.vortex. The floating point programs (CFP95) are 101.tomcatv, 102.swim, 103.su2cor, 104.hydro2d, 107.mgrid, 110.applu, 125.turb3d, 141.apsi, 145.fpppp, and 146.wave5. A detailed description of the benchmarks can be found in the SPEC newsletters [98]. For each benchmark, I tabulate the inputs used for the experiments and the inputs for profiling in Table 5-2 and Table 5-3**.**

5.3 Experiments

In one set of experiments, I study the effect of compiler task partitioning heuristics and task characteristics of Multiscalar programs. In another set of experiments, I study the effectiveness of compiler strategies for register communication and compiler scheduling to streamline register communication. I also measure the impact of dead register analysis on register communication traffic.

Since the number of instructions executed by a program changes due to compiler techniques, all relevant quantities have been scaled appropriately. Register communication scheduling was performed in all the experiments measuring the effectiveness task selection heuristics and task characteristics. Data dependence tasks were used in all the experiments measuring register communication strategies and scheduling. This combination was used so that results are not skewed by unoptimized code.

To isolate the effect of the different task selection heuristics, I present measurements based on four types of tasks: (1) Basic block tasks which are obtained by demarcating each basic block to a task on its own (i.e., each task contains a single basic block), (2) control flow tasks which are obtained by including multiple basic blocks without exceeding

Table 5-2: Inputs for CINT benchmarks. The column titled input shows the input used to run the benchmark for each experiment. The column titled profile shows the input used to profile the benchmark. Note that in some cases the SPEC standard test inputs are used to profile and the SPEC standard train inputs are used to run, to keep the simulation runs to fewer than many billion instructions.

Benchmarks	Input	Profile	Description
099.go	2stone9	null	Plays the game of go against itself, great deal of pattern matching and look-ahead logic, cache activity is small.
124.m88ksim	dhry	dcrand	A clock-level simulator (includes cache latencies) for the Motorola 88100, known to be not vectorizable.
126.gcc	cccp	jump	A GNU C compiler version 2.5.3, converts preprocessed files to optimized Sparc assembly, known to be not vectorizable.
129.compress	train	test	Reduces the size of the named files using adaptive Lempel-Ziv coding, modified to work out of memory instead of using files, sensitive to data cache size.
130.li	train	test	A Lisp interpreter written in C, known to be not vectorizable.
132.ijpeg	vigo	specmu	Performs image compression/decompression on in-memory images based on the JPEG facilities, modified to work out of memory instead of using files.
134.perl	jumble	scrabbl	An interpreter for the Perl language, inputs are scripts that perform some basic math calculations and word lookups in associative arrays, may spend as much as 10% of total time in library routines like malloc, free, memcpy, etc.
147.vortex	train	test	A subset of a full object-oriented database program called VORTEx, builds and manipulates three separate, but inter-related databases, database size restricted to 40MB, has not been vectorized as of SPEC95 release.

four targets, as described in Section 3.4.2, (3) data dependence tasks which are obtained by including data dependence chains (mostly register dependences) combined with con-

Table 5-3: Inputs for CFP benchmarks. The column titled input shows the input used to run the benchmark for each experiment. The column titled profile shows the input used to profile the benchmark. Note that in some cases the SPEC standard test inputs are used to profile and the SPEC standard train inputs are used to run, to keep the simulation runs to fewer than many billion instructions.

Benchmarks	Input	Profile	Description
101.tomcatv	test	train	A vectorizable mesh generation program, sensitive to memory subsystem performance, parallelizable.
102.swim	test	train	A shallow water model with a 1024 x 1024 grid of finite difference approximations, vectorizable and parallelizable.
103.su2cor	test	train	Computes masses of elementary particles applying a monte carlo method, vectorizable and parallelizable.
104.hydro2d	test	train	Solves hydrodynamical Navier Stokes equations to compute galactical jets, vectorizable and parallelizable.
107.mgrid	test	train	A multi-grid solver in 3D potential field, parallelizable.
110.applu	train	test	A parabolic/elliptic partial differential equations solver, parallelizable.
125.turb3d	test	train	Simulates isotropic, homogeneous turbulence in a cube, uses a large 1D FFT computation, parallelizable.
141.apsi	train	test	Solves problems regarding temperature, wind, velocity, and distribution of pollutants, not easily parallelizable.
145.fpppp	train	test	Solves an integral derivative that models interactions among a number of atoms, minimal cache footprint, not easily parallelizable.
146.wave5	train	test	Solves Maxwell’s equations on a cartesian mesh, considerable indirect addressing used, not easily parallelizable.

trol flow heuristic, as described in Section 3.4.3, and (4) task size heuristic tasks which are obtained by including entire function invocations (which execute fewer than 15 instructions) within tasks.

5.3.1 Effectiveness of compiler heuristics

In order to study the effectiveness of the heuristics, I used basic block tasks as the base case. Basic block tasks are small and incur performance loss from all quarters. All branches are exposed to the prediction hardware without taking advantage of reconvergent control flow paths. Data dependencies (register) from one basic block to its successor are dense and cause long communication delays. Memory data misspeculations do not occur because not many memory dependencies are speculated. Overheads are significant because of the small size of basic block tasks. The window of instructions established by basic block tasks is not large enough to expose enough parallelism.

The heuristics are able to alleviate the problems of basic block tasks to a certain extent by establishing a larger window. Heuristic tasks are larger and more independent because they include more control and data dependencies within them instead of exposing them.

The heuristics capture both regular and irregular parallelism. For the integer benchmarks, tasks obtained using the heuristics range from a few basic blocks to entire function invocations. Let us consider each benchmark separately to understand its behavior. 129.compress and 134.perl contain data dependence recurrences in the frequently executed parts of the programs causing the heuristics to split loops into multiple tasks. Consequently, 129.compress and 134.perl aggravate control and data dependencies. 130.li contains a high frequency of recursive function invocations. If included within tasks, the function calls cause load imbalance and otherwise, cause function bodies to be split into small tasks, exacerbating inter-task data dependencies and overheads. 126.gcc contains dense, irregular data dependencies which are hard to include within tasks because they are numerous. 126.gcc also incurs performance loss due to large number of function calls like 130.li. 132.ijpeg contains abundant regular loop-parallelism which the heuristics exploit although the loops access memory via pointers. 124.m88ksim and 147.vortex contain irregular parallelism that can be captured by non-loop tasks.

For the floating point benchmarks, the heuristics consistently exploit loop-parallelism. Most of the tasks are loop bodies and any loop-recurrences are synchronized by the hardware [73] avoiding data dependence misspeculations. The fine grain synchronization is overlapped with intra-iteration parallelism. Even basic block tasks perform well attesting to the high level of parallelism in floating point programs.

Figure 5-1 shows the improvements in IPC using control flow heuristic, data dependence heuristic, and task size heuristic for out-of-order PU configurations executing the integer benchmarks over the base case of basic block tasks. Figure 5-2 shows the improvements in IPC using control flow heuristic, data dependence heuristic, and task size heuristic for out-of-order PU configurations executing the floating point benchmarks over the base case of basic block tasks.

The compiler heuristics (control flow, data dependence and task size together) are effective in capturing parallelism beyond basic block tasks. Using out-of-order PUs, the integer benchmarks improved by 19%-38% and 25%-39% on 4 and 8 PUs, respectively, over basic block tasks (Figure 5-1), while the floating point benchmarks were boosted by 21%-52% and 25%-53% on 4 and 8 PUs, respectively, over basic block tasks (Figure 5-2). The floating point benchmarks have more regular parallelism than the integer benchmarks, as a result of which the heuristics succeed in extracting more parallelism from the floating point benchmarks.

For the integer benchmarks, control flow heuristic improves performance 23%-54% and 23%-53% for 4 and 8 out-of-order PU configurations, respectively, over basic block tasks

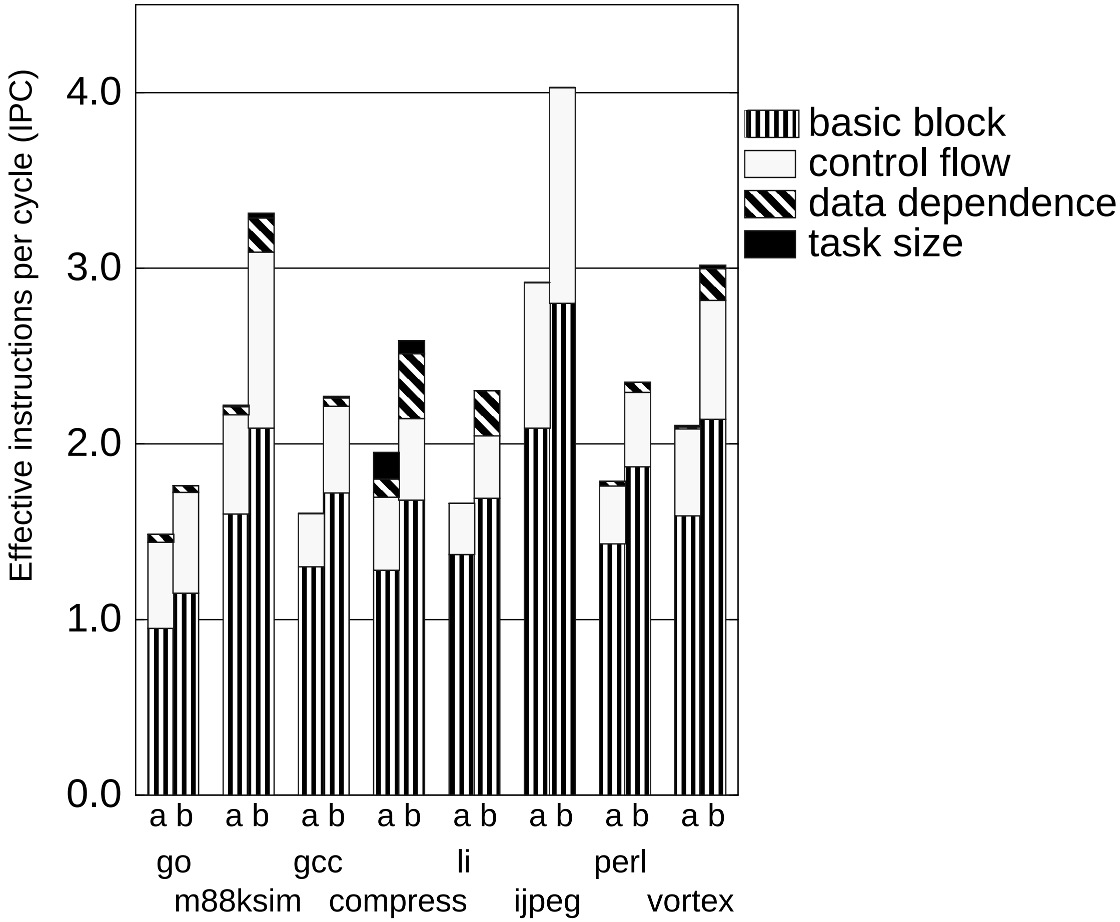


Figure 5-1: Impact of compiler heuristics on CINT benchmarks with out-of-order PUs. The two experiments marked a and b use 4 and 8 out-of-order PUs, respectively.

(Figure 5-1). It is important to note that the measurements shown here for data dependence heuristic are over and above control flow heuristic, i.e., data dependence heuristic is applied in conjunction with control flow heuristic. Data dependence heuristic adds modest performance improvements (<1%-6% and <1%-15% for 4 and 8 PUs, respectively) over control flow heuristic. There are many reasons for the improvements being modest: (1) Out-of-order PUs can tolerate latencies due to register communication delays significantly and (2) By including adjacent basic blocks within a task, control flow heuristic already

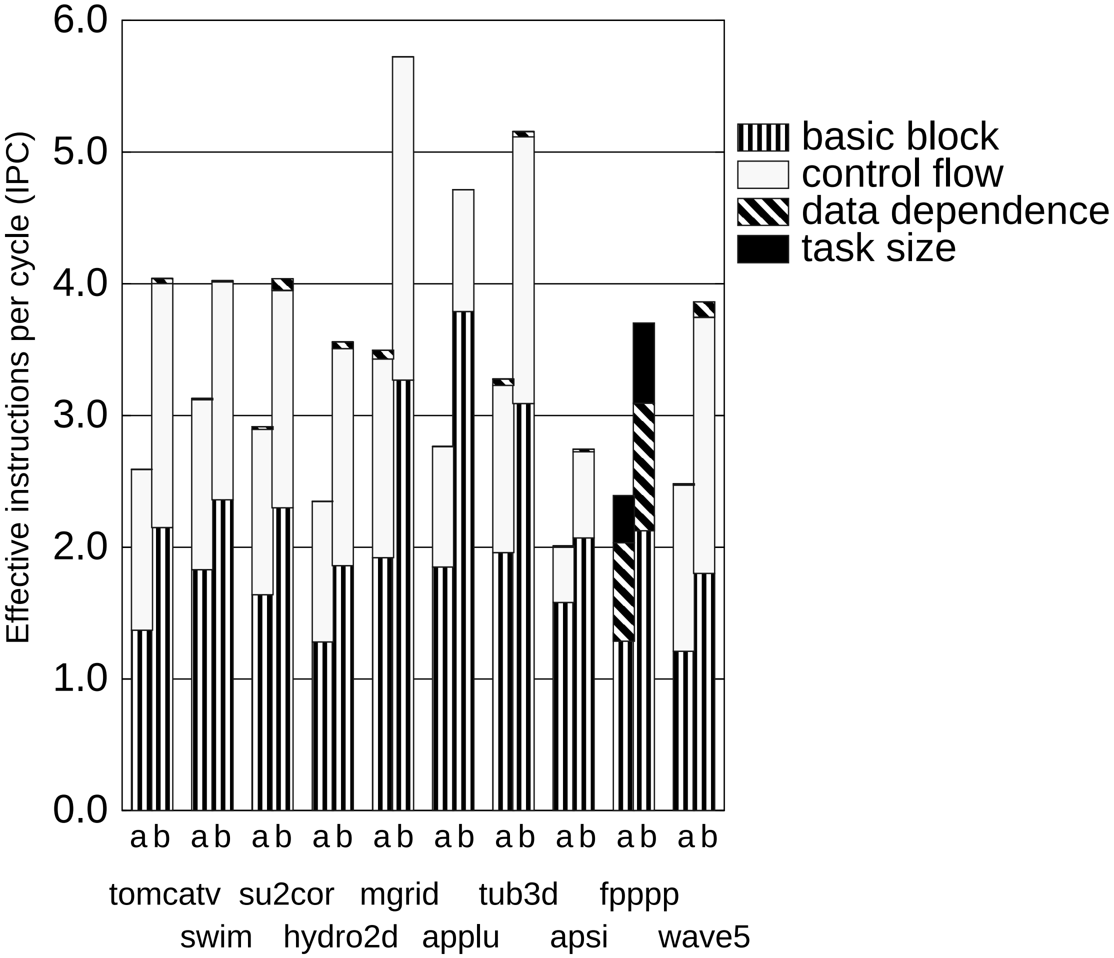


Figure 5-2: Impact of compiler heuristics on CFP benchmarks with out-of-order PUs. The two experiments marked a and b use 4 and 8 out-of-order PUs, respectively.

includes data dependence chains within tasks; data dependence heuristic has fewer opportunities to further capture data dependences.

Figure 5-3 shows the impact of control flow heuristic, data dependence heuristic, and task size heuristic on the integer benchmarks for in-order PUs. Put together, the heuristics improve performance by 23%-70% and 31%-83% for 4 and 8 PU configurations over basic block tasks. These improvements are better than those for out-of-order PUs because

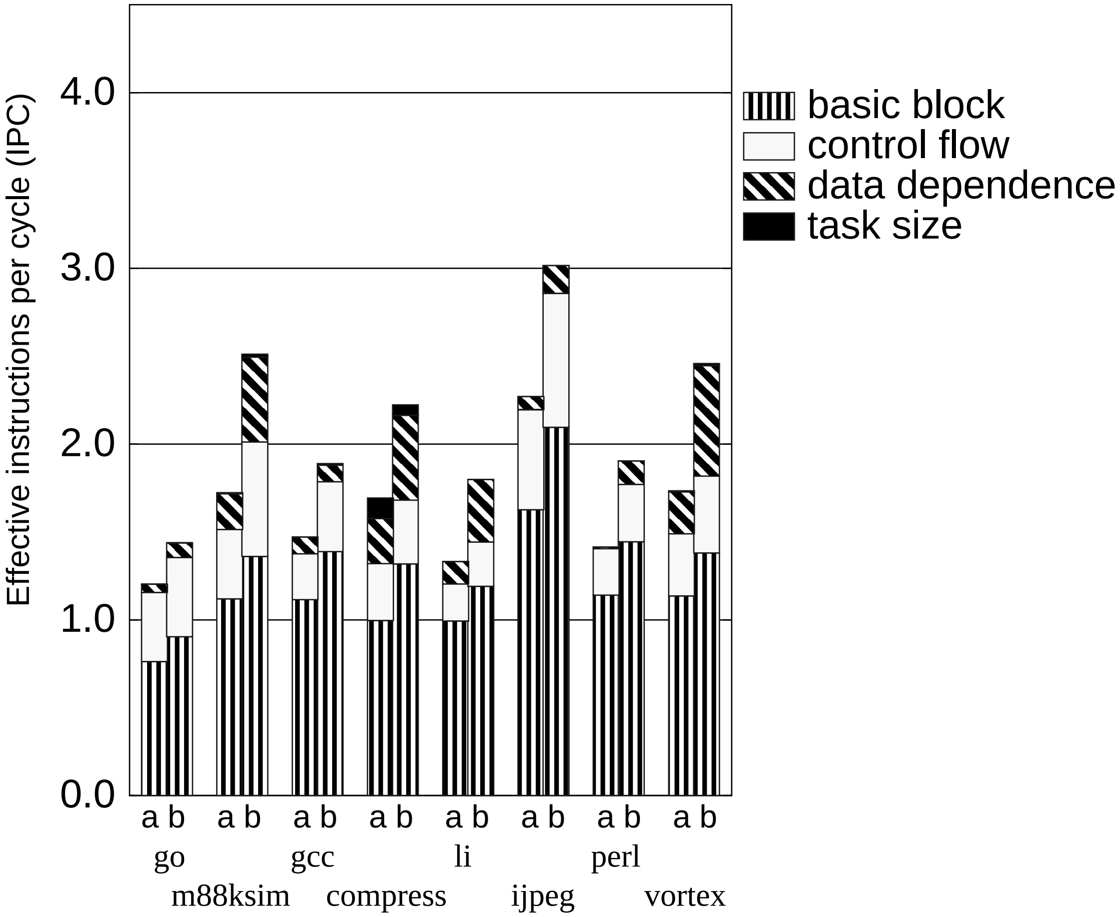


Figure 5-3: Impact of compiler heuristics on CINT benchmarks with in-order PUs. The two experiments marked a and b use 4 and 8 in-order PUs, respectively.

in-order PUs do not have as much latency tolerance as out-of-order PUs; the heuristics are effective in avoiding inter-task dependences, which stifle the in-order PUs more than the out-of-order PUs. Using in-order PUs, data dependence heuristic improves performance by 3%-15% and 5%-25% for 4 and 8 PU configurations, respectively, over control flow heuristic. Out-of-order PUs can tolerate more latencies than in-order PUs. As expected, data dependence heuristic improves performance more for in-order PU configurations than for out-of-order configurations. For data dependence tasks, 8 in-order PUs consistently perform better than 4 out-of-order PUs[3].

For the floating point benchmarks, control flow heuristic improves performance by 24%-106% and 24%-117% for 4 and 8 out-of-order PU configurations, respectively, over basic block tasks (Figure 5-2). Impact of data dependence heuristic is less pronounced because control flow heuristic generates tasks which exploit loop-level parallelism. This trend is true for in-order PUs as well. Figure 5-4 shows the impact of control flow heuristic, data dependence heuristic, and task size heuristic on the floating point benchmarks for in-order PU configuration.The only exception to exploiting loop-level parallelism is 145.fpppp, where data dependence heuristic improves performance significantly. Put together, the heuristics improve performance by 29%-106% and 24%-114% for 4 and 8 PU configurations over basic block tasks. For control flow tasks, 8 in-order PUs consistently perform better than 4 out-of-order PUs.

Task size heuristic improves performance of 129.compress by including short (5 instructions) function invocations in the most frequent loop of the program and 145.fpppp by splitting large (~1000 instructions) basic blocks into smaller basic blocks. Task size heuristic does not impact the other benchmarks due to lack of opportunity.

5.3.2 Task characteristics and metrics

I now present measurements of overall performance various task characteristics for tasks selected by the different heuristics. First, I measure performance achieved by tasks selected by the different heuristics. I measure (1) task size in terms of dynamic instructions, (2) inter-task control flow, and (3) inter-task data communication and speculation. Task size heuristic affects only two of the benchmarks: 129.compress and 145.fpppp. So, I include task size heuristic only for these benchmarks in the analyses of task characteristics.

[3] An important point to note is that 8 PUs have more bandwidth to the memory subsystem than 4 PUs.

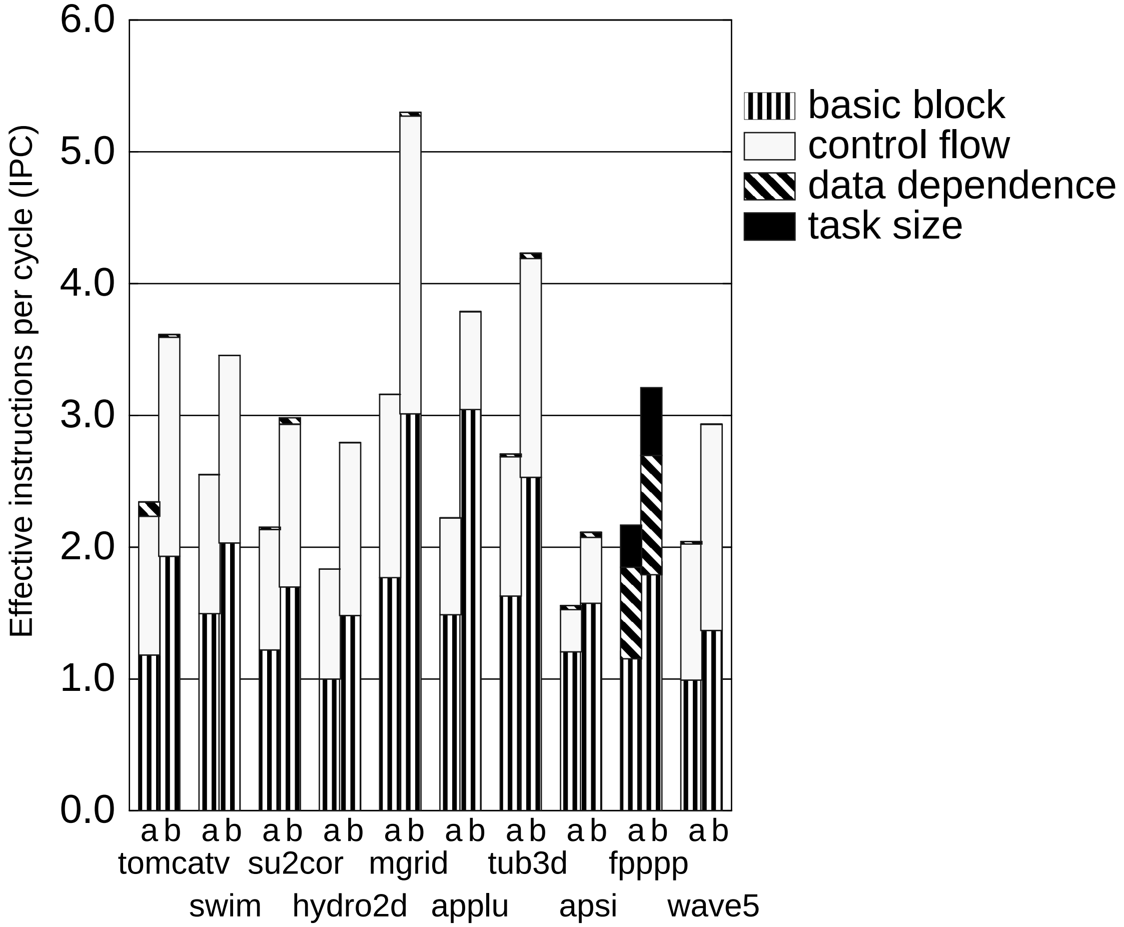


Figure 5-4: Impact of compiler heuristics on CFP benchmarks with in-order PUs. The two experiments marked a and b use 4 and 8 in-order PUs, respectively.

5.3.2.1 Task size metrics

To characterize tasks further, I measure the total number of dynamic instructions and the number of control transfer instructions (i.e., branches, jumps, indirect jumps, function calls, and function returns) executed in a task. While the total is a measure of task size, the control transfer count measures the number of basic blocks included in control flow tasks and data dependence tasks. I also present the number of cycles spent by an instruction due to load imbalance.

In Table 5-4 and Table 5-5, I present the average dynamic size of basic block tasks, control flow tasks and data dependence tasks for the integer and floating point benchmarks.

Table 5-4: Dynamic task size for CINT benchmarks. Columns titled Basic Block Tasks, Control Flow Tasks, and Data Dependence Tasks show task sizes in number of dynamic instructions for the corresponding heuristic. Columns titled total show the number of dynamic instructions and columns titled control show the number of dynamic control transfer instructions per task. Since 129.compress responds to task size heuristic, both control flow tasks and data dependence tasks are augmented with task size heuristic for this benchmark.

Benchmarks	Basic Block	Control Flow		Data Dependence	
	total	control	total	control	total
099.go	6.4	2.53	18.2	2.04	12.7
124.m88ksim	4.3	2.97	14.8	2.42	10.3
126.gcc	5.8	2.52	12.4	2.32	11.6
129.compress*	5.7	1.78	10.2	2.77	15.0
130.li	3.9	1.89	8.1	1.64	7.1
132.ijpeg	10.6	2.42	23.3	2.43	23.8
134.perl	6.5	2.26	14.9	2.20	10.6
147.vortex	6.9	2.41	17.2	2.19	14.0

For the integer benchmarks, basic block tasks contain fewer than 10 instructions (Table 5-4). Control flow tasks and data dependence tasks contain usually more than twice as many instructions as basic block tasks. For the floating point benchmarks, even basic block tasks contain more than 20 instructions, except for 107.hydro2d (Table 5-5). Similar to their integer counterparts, control flow and data dependence tasks of floating point programs contain usually more than twice as many instructions as basic block tasks. The main reason for data dependence tasks to contain fewer dynamic instructions than control flow tasks is that control flow heuristic is greedy and includes basic blocks past data dependence chains, whereas data dependence heuristic terminates tasks as soon as a data depen-

Table 5-5: Dynamic task size for CFP benchmarks. Columns titled Basic Block, Control Flow, and Data Dependence show task size for the corresponding heuristic. Columns titled control show the number of dynamic instructions and columns titled control show the number of dynamic control transfer instructions per task. Since 145.fpppp responds to task size heuristic, both control flow tasks and data dependence tasks are augmented with task size heuristic for this benchmarks.

Benchmarks	Basic Block total	Control Flow		Data Dependence	
		control	total	control	total
101.tomcatv	44.1	4.96	114.9	3.24	84.8
102.swim	42.0	4.13	87.7	4.13	87.7
103.su2cor	49.8	8.03	107.8	8.03	107.8
104.hydro2d	11.9	6.0	44.0	5.24	39.5
107.mgrid	51.4	2.0	105.5	2.01	107.4
110.applu	21.7	1.71	39.0	1.70	38.5
125.turb3d	21.2	2.46	41.7	2.44	40.8
141.apsi	24.8	2.78	51.0	2.63	46.8
145.fpppp*	957.8	1.45	59.0	2.50	66.5
145.wave5	24.4	4.20	59.1	4.08	56.1

dence is included. 129.compress, 107.mgrid, 144.fpppp do not follow this trend because data dependence tasks steer task selection to paths different from control flow tasks, resulting in entirely different tasks.

A few assembly-level macro instructions expand into a sequence of instructions containing control transfer instructions. For example, an integer divide macro expands into a sequence of instructions that includes a branch to check for divide by zero. Several floating point programs use library calls for various computations (e.g., trigonometric functions). So, some basic block tasks may include such control transfer instructions which are hidden from the compiler. Also, control flow tasks and data dependence tasks include extra jumps from loop restructuring optimization. For these reasons, there may seem to be a discrepancy between ratio of the size of basic block tasks and that of heuristic tasks and the number of control transfer instructions in heuristic tasks. Control flow and data dependence tasks contain between 2 and 3 basic blocks per task as indicated by the number of control transfer instructions (Table 5-4 and Table 5-5). The number of control transfer instructions contained in control flow tasks and data dependence tasks is larger than their integer counterparts.

While the tasks of the integer programs are fairly small, the tasks of the floating point programs are reasonably large due to high frequency loops that do not contain any function invocations. In the integer benchmarks, function invocations often result in small tasks unless entire function calls are included within tasks. Task size heuristic includes short function calls within tasks. Including large function calls or recursive calls lead to inordinate number of memory dependence squashes and excessive load imbalance. Large tasks also fill up the ARB. Setting the threshold level to include entire function invocations within tasks at fewer than 15 instructions avoids these problems. 129.compress has two function invocations that execute 5 dynamic instructions in the most frequent loop of the program. Among the integer benchmarks, only 129.compress responds to task size heuristic. Apart from function invocations, loops with a few iterations and short loops with tight recurrences lead to small tasks in the integer benchmarks.

In order to isolate the effect of each of the performance issues such as load imbalance, task overhead, inter-task register dependences and inter-task memory dependences, I present the cycles spent due to each of these issues in some of the following sections of this chapter. Since out-of-order PUs can overlap delays, these measurements were obtained by monitoring the commit point of the PU. If the instruction at the commit point waits for register values and no other useful work is done in that cycle, then the cycle is charged to register communication delay. This method gives only an upper bound on the number cycles spent and not the exact cycle count accountable to each performance issue separately because there could be other instructions, whose delay is overlapped with the instruction at the commit point. For example, if register communication delay is measured to be 20% of overall execution time, performance may not improve by 12.5% if register communication is completely removed because there may be other instructions waiting due to memory dependences overlapped with the instructions waiting for register values. Since these measurements do not correspond to exact, isolated performance components, I present the “observed stall” in CPI instead of IPC to avoid any confusion.

In Figure 5-5 and Figure 5-6, I present the average number of cycles spent by an instruction due to load imbalance for the integer and the floating point benchmarks, respectively. Control flow tasks and data dependence tasks exhibit less load imbalance for both the integer and the floating point benchmarks, as compared to basic block tasks. This disparity is due to the amortization of load imbalance loss over larger number of instructions of control flow tasks and data dependence tasks. Control flow tasks are a little worse than data dependence tasks due to the greedy nature of control flow heuristic. Control flow heuristic tends to create small tasks when it encounters terminal basic blocks and terminal control edges, while greedily creating larger tasks with non-terminal basic blocks and non-terminal edges; this tendency causes disparity in task size, whereas data dependence heuristic terminates tasks as soon as a data dependence chain is included, distributing basic blocks more evenly among control flow paths. Basic block tasks in 129.compress and 132.ijpeg incur significantly high load imbalance because high-frequency loops contains varied-sized basic blocks.

In the case of the floating point programs, many basic blocks contain substantial amounts of computation in high-frequency loops. But basic blocks containing a few induction variable increments in those loops contribute to the disparity in the amount of computation performed per basic block. Basic block tasks incur high load imbalance due to this reason (Figure 5-6). For control flow tasks and data dependence tasks, load imbalance arises due to variation in the amount of computation done in loop iterations.

Increasing the number of PUs is expected to increase load imbalance because larger number of tasks have a greater probability of being imbalanced. For example, increasing

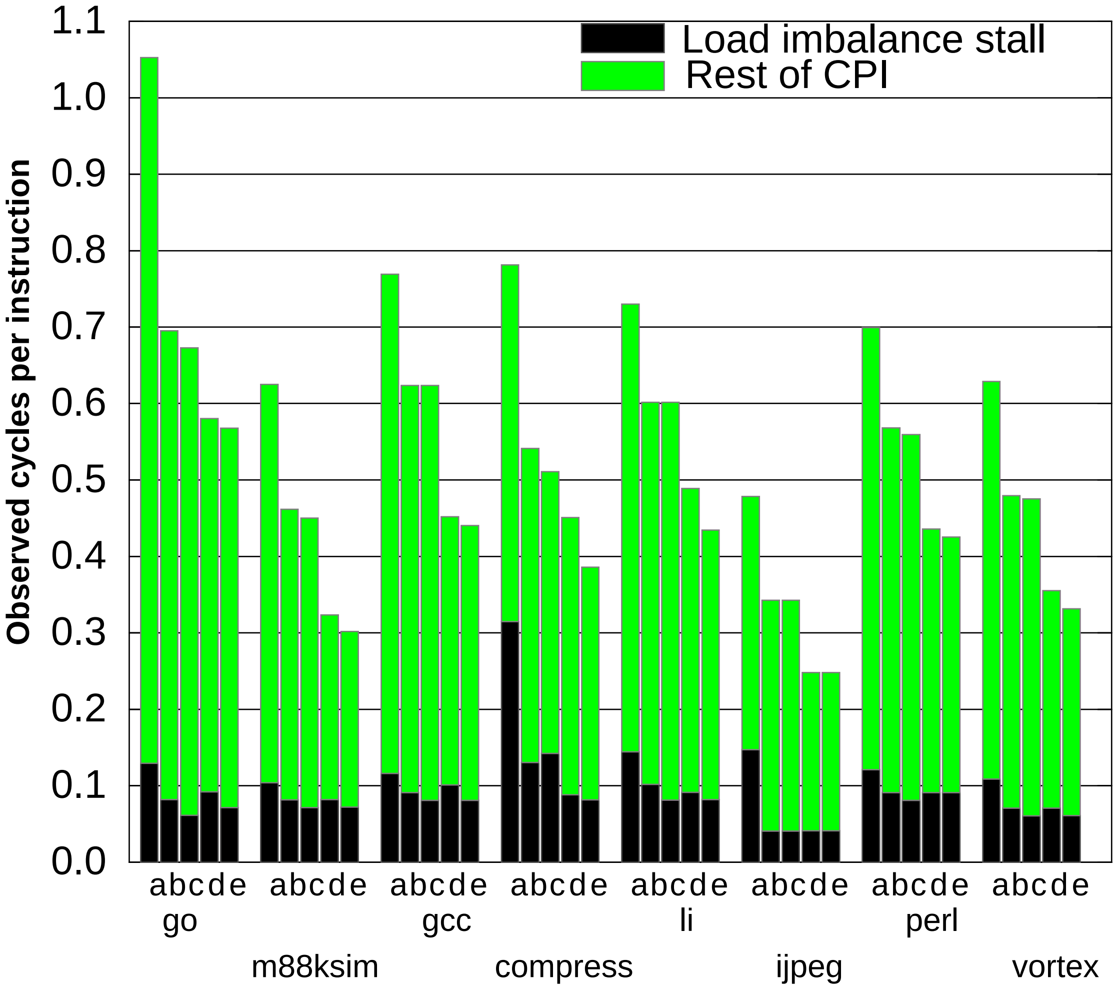


Figure 5-5: Observed load imbalance stall for SPEC CINT benchmarks. All the experiments use out-of-order PUs. The five experiments marked a, b, c, d, and e are basic block tasks, control flow tasks on 4 PUs, data dependence tasks on 4 PUs, control flow tasks on 8 PUs, and data dependence tasks on 8 PUs, respectively. For 129.compress, control flow tasks and data dependence tasks are augmented with task size heuristic.

the number of PUs from 4 to 8 exposes the variation in size of tasks that are 4, 5, 6, or 7 dynamic tasks apart. But Figure 5-11 and Figure 5-12 show that the average time spent by an instruction due to load imbalance in a 8-PU configuration is not much less than that in a 4-PU configuration for most benchmarks. The main reason for the similarity in load imbalance is that variation in adjacent dynamic tasks contribute to overall load imbalance

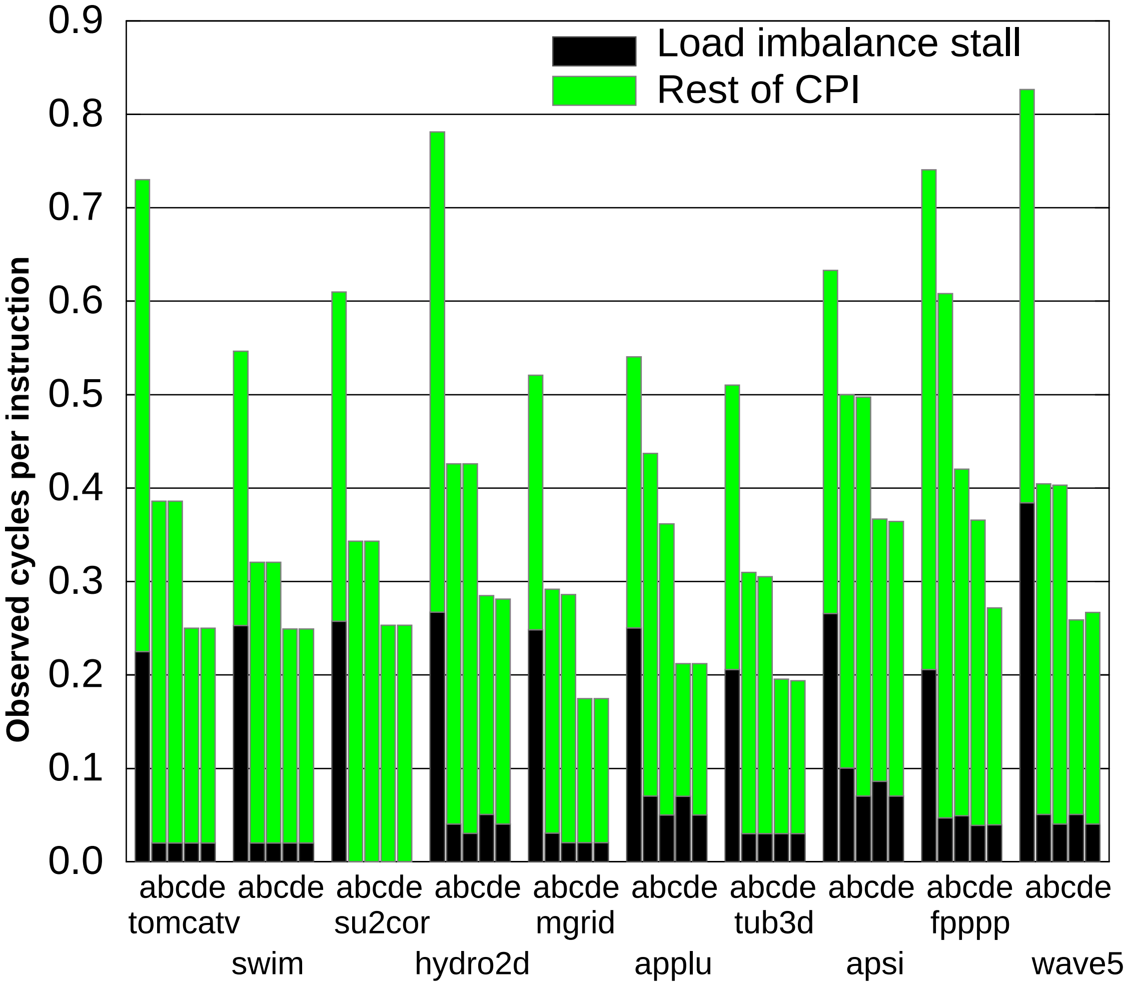


Figure 5-6: Observed load imbalance stall for SPEC CFP benchmarks. All the experiments use out-of-order PUs. The five experiments marked a, b, c, d, and e are basic block tasks, control flow tasks on 4 PUs, data dependence tasks on 4 PUs, control flow tasks on 8 PUs, and data dependence tasks on 8 PUs, respectively. For fpppp, control flow tasks and data dependence tasks are augmented with task size heuristic.

more than variation in dynamic tasks that are farther apart. Since the overall CPI decreases on increasing the number of PUs (up to some number of PUs), load imbalance becomes a more important performance factor, as a percentage of overall CPI, for larger number of PUs. For control flow tasks load imbalance accounts for 9%-16% and 12%-25% of total execution time for 4-PU and 8-PU configurations, respectively. For data dependence tasks, load imbalance accounts for 9%-16% and 12%-24% of total execution time for 4-PU and 8-PU configurations, respectively.

In Figure 5-7 and Figure 5-8, I present the average number of cycles spent by an instruction due to task overhead for the integer and the floating point benchmarks, respectively. Task overhead decreases significantly for control flow tasks and data dependence tasks over basic block tasks due to their larger size, for the integer benchmarks. Since even basic block tasks are large for the floating point benchmarks, the programs incur little overhead. Increasing the number of PUs decreases task overhead because of overlap of overhead time among more PUs.

5.3.2.2 Inter-task control flow metrics

To characterize inter-task control flow, I measure control flow prediction accuracies for tasks obtained by the different heuristics and the distribution of dynamic targets taken. In Table 5-6 and Table 5-7, I present inter-task control flow misprediction rates of basic block tasks, control flow tasks and data dependence tasks. Prediction accuracy of basic block tasks is higher than that of superscalar branch prediction accuracy because it includes branches, jumps, function calls and returns. In general, the prediction hardware is able to maintain high task prediction accuracies for control flow tasks and data dependence tasks (column Task) in spite of predicting one of four targets, where as basic block tasks expose only two targets.

Since control flow tasks and data dependence tasks usually contain multiple branches per task (shown in Table 5-6 and Table 5-7), comparing prediction accuracies of these tasks with those of basic block tasks requires normalizing the accuracies with respect to the average number of dynamic branches per task. To this end, I present two kinds of prediction accuracies for control flow tasks and data dependence tasks (columns Task and Branch): (1) Misprediction accuracy calculated as a ratio of the number of incorrect task

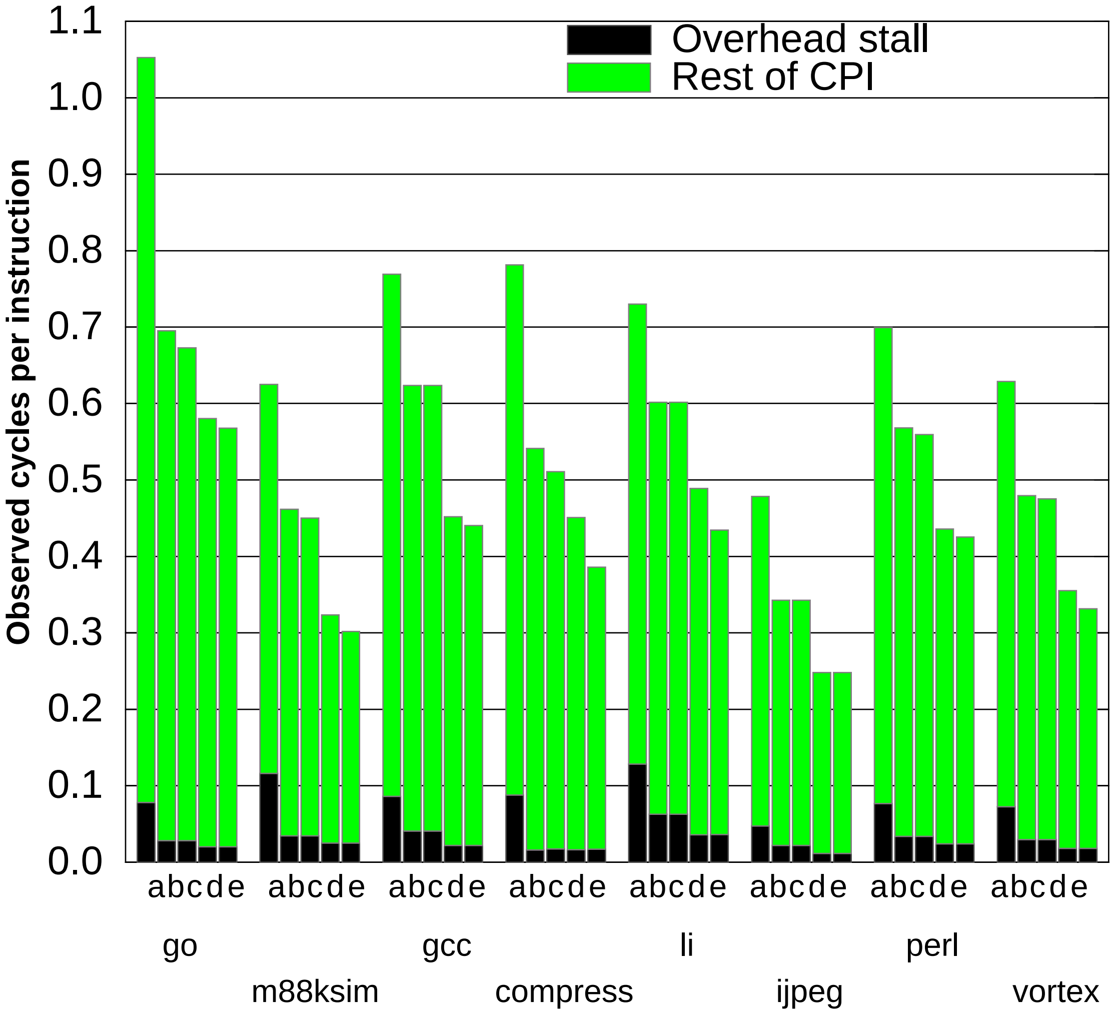


Figure 5-7: Observed task overhead stall for SPEC CINT benchmarks. All the experiments use out-of-order PUs. The five experiments marked a, b, c, d, and e are basic block tasks, control flow tasks on 4 PUs, data dependence tasks on 4 PUs, control flow tasks on 8 PUs, and data dependence tasks on 8 PUs, respectively. For 129.compress, control flow and data dependence tasks are augmented with task size heuristic.

predictions and the number of all task predictions. (2) Misprediction accuracy calculated as a ratio of the number of incorrect task predictions and the number of dynamic branches.

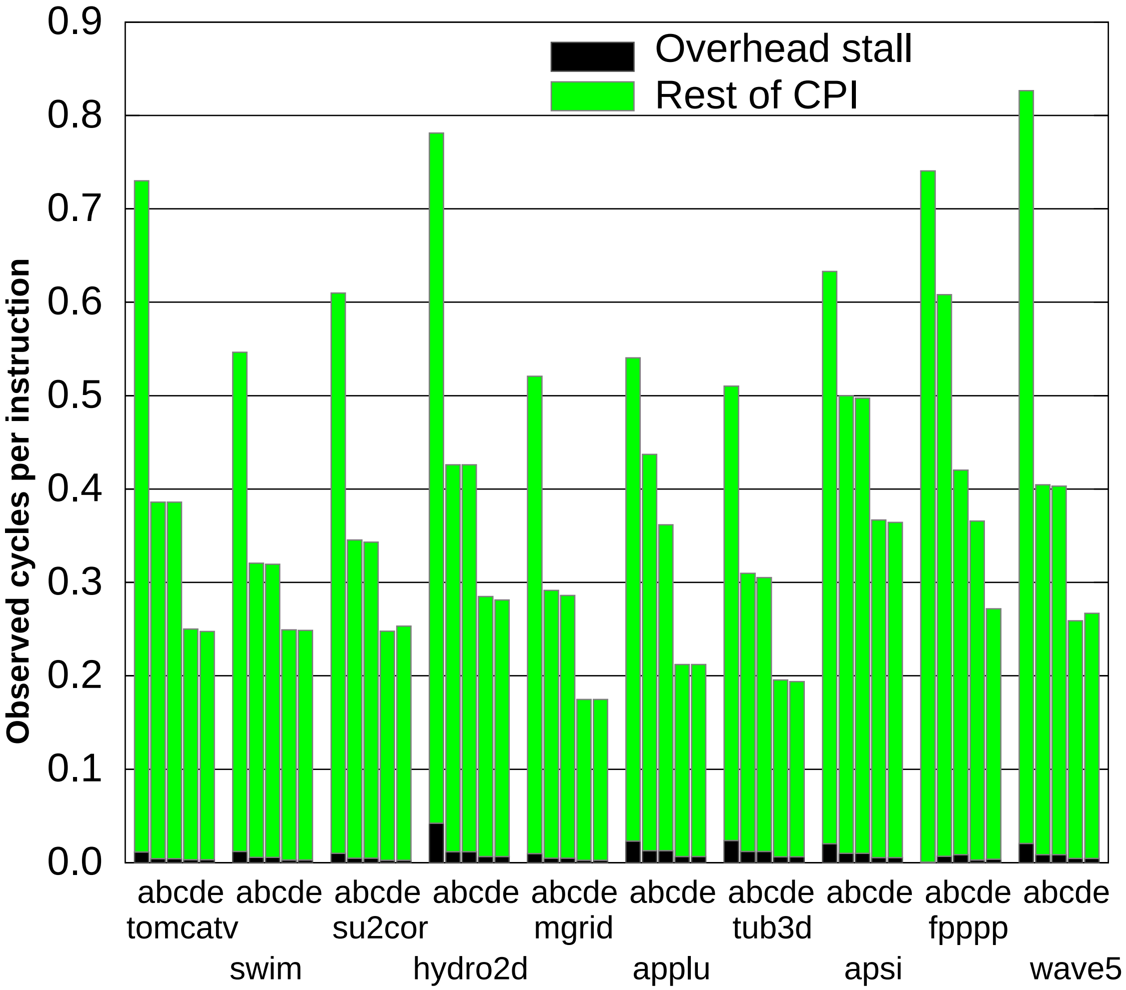


Figure 5-8: Observed task overhead stall for SPEC CFP benchmarks. All the experiments use out-of-order PUs. The five experiments marked a, b, c, d, and e are basic block tasks, control flow tasks on 4 PUs, data dependence tasks on 4 PUs, control flow tasks on 8 PUs, and data dependence tasks on 8 PUs, respectively. For fpppp, control flow and data dependence tasks are augmented with task size heuristic.

Comparing basic block tasks with control flow tasks in terms of task prediction accuracies (column Task), there are two kinds of behavior: Task prediction accuracies are higher for control flow tasks than basic block tasks for those benchmarks which capture loop-level tasks, namely, 132.ijpeg, 101.tomcatv, 102.swim, 103.su2cor, 104.hydro2d, 107.mgrid, and 145.wave5. In these benchmarks, the most frequent tasks are loop bodies

Table 5-6: Control Flow Misspeculation Rate for SPEC95 CINT Benchmarks. Columns titled Basic Block Tasks, Control Flow Tasks, and Data Dependence Tasks show misprediction rates for the corresponding heuristics. Columns titled Task show the task prediction accuracies and columns titled Branch show the effective prediction accuracy normalized with respect to the average number of branches per task. Since 129.compress responds to task size heuristics, both control flow and data dependence tasks are augmented with task size heuristics for this benchmark.

Benchmarks	Basic Block Tasks	Control Flow Tasks		Data Dependence Tasks	
		Task	Branch	Task	Branch
099.go	14.4%	14.7%	5.8%	14.6%	7.2%
124.m88ksim	3.1%	4.0%	1.4%	4.9%	2.0%
126.gcc	4.4%	5.8%	2.3%	7.4%	3.2%
129.compress*	5.0%	5.7%	3.2%	7.8%	2.8%
130.li	3.3%	4.0%	2.1%	5.2%	3.2%
132.ijpeg	6.0%	3.7%	1.5%	5.1%	2.1%
134.perl	2.1%	3.9%	1.7%	4.1%	1.9%
147.vortex	0.8%	0.7%	0.3%	0.7%	0.3%

which do not expose any of the branches internal to the loop bodies and are easy to predict. Data dependence tasks have worse task prediction accuracies than control flow task because including data dependence chains within tasks is preferred over reconvergent control flow paths or loop bodies.

If task prediction accuracy is normalized over the number of dynamic branches (column Branch), the effective prediction accuracies are significantly better for control flow tasks and data dependence tasks, demonstrating the synergy between the heuristics and control flow speculation hardware. Although the normalized accuracies are higher, an important caveat is that average misspeculation penalties are larger for heuristic tasks than for basic block tasks due to their larger sizes.

Table 5-7: Control flow misspeculation rate for SPEC95 CFP benchmarks. Columns titled Basic Block Tasks, Control Flow Tasks, and Data Dependence Tasks show misprediction rates for the corresponding heuristics. Columns titled Task show the task prediction accuracies and columns titled Branch show the effective prediction accuracy normalized with respect to the average number of branches per task. Since 145.fpppp responds to task size heuristics, both control flow and data dependence tasks are augmented with task size heuristics for this benchmark.

Benchmarks	Basic Block Tasks	Control Flow Tasks		Data Dependence Tasks	
		Task	Branch	Task	Branch
101.tomcatv	1.6%	0.4%	0.1%	0.4%	0.1%
102.swim	0.1%	0.2%	0.0%	0.2%	0.0%
103.su2cor	3.4%	0.5%	0.1%	0.5%	0.1%
104.hydro2d	0.1%	0.3%	0.1%	0.2%	0.0%
107.mgrid	1.1%	2.2%	1.1%	2.2%	1.1%
110.applu	3.9%	3.9%	2.3%	4.2%	2.5%
125.turb3d	3.4%	5.8%	2.4%	6.7%	2.7%
141.apsi	2.9%	4.3%	1.5%	4.1%	1.6%
145.fpppp*	5.6%	1.8%	1.2%	2.8%	1.1%
145.wave5	0.8%	0.8%	0.2%	1.1%	0.3%

In Figure 5-9 and Figure 5-10, I present the distribution of dynamic tasks with respect to number of targets. The measurements were made on data dependence tasks because they achieve the highest performance. Almost 60% and 90% of dynamic tasks have two or fewer targets, for the integer and the floating point benchmarks, respectively. For the integer benchmarks, the average number of dynamic targets is mostly between 2 and 3, indicating that although multiple basic blocks are included in a task, the heuristics are able to take advantage of reconvergent control flow paths. 132.ijpeg has unusually large number of two-target tasks, due to many loop body tasks, which have two targets, namely, the loop back and the fall through. For the floating point benchmarks, the average number of dynamic targets is mostly close to 2 because of loop body tasks.

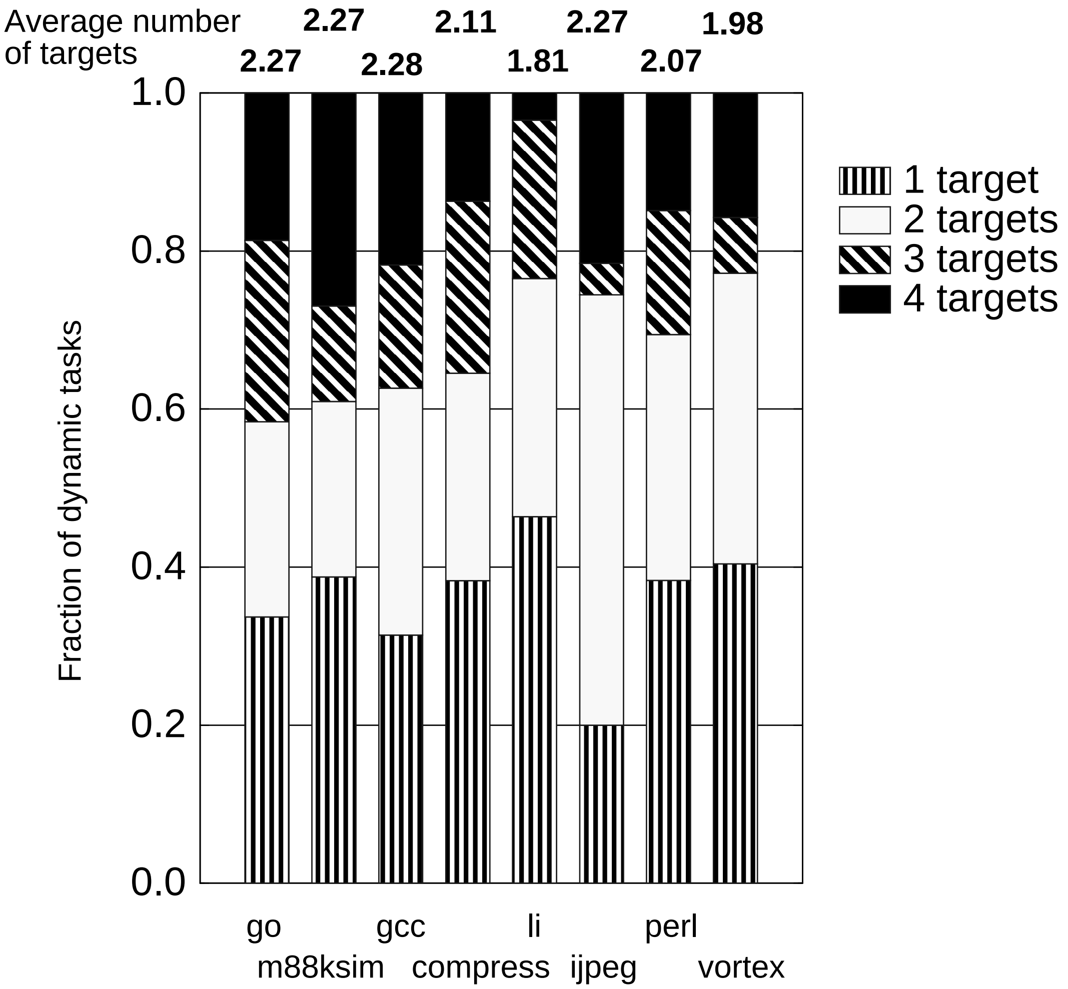


Figure 5-9: Number of targets for SPEC95 CINT benchmarks. Distribution of dynamic data dependence tasks as a function of the number of targets.

5.3.2.3 Inter-task data dependence metrics

To characterize inter-task data dependences, I measure the average number of cycles spent by an instruction waiting for register values and memory values and the fraction of inter-task data dependences in the program as a function of the number of dynamic tasks that the data dependences span. In Figure 5-11 and Figure 5-12, I present the average number of cycles spent by an instruction waiting for register values for the integer and

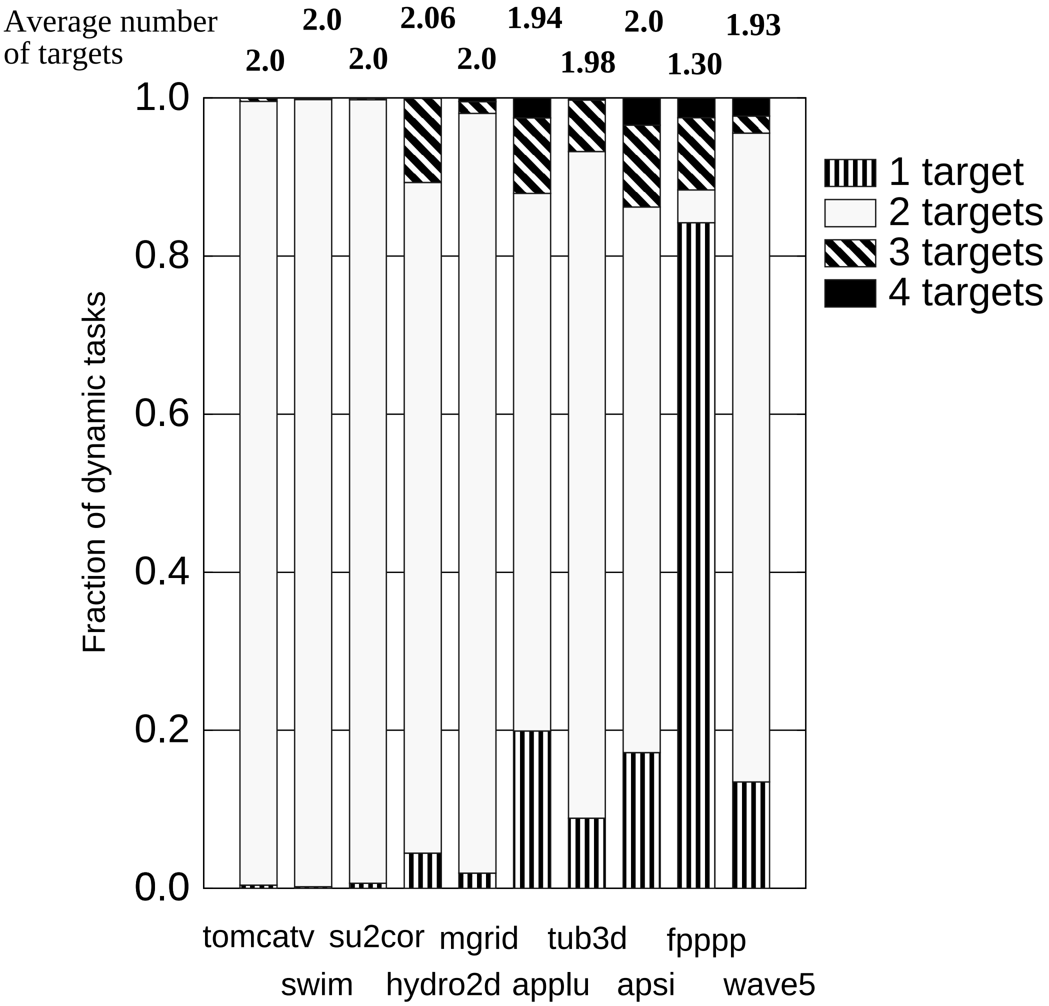


Figure 5-10: Number of targets for SPEC95 CFP benchmarks. Distribution of dynamic data dependence tasks as a function of the number of targets.

floating point benchmarks, respectively. In general, floating point benchmarks incur fewer cycles waiting for register communication than integer benchmarks because floating point tasks are mostly loop bodies and there are few loop-carried register value dependences in these benchmarks. I analyze the effect of increasing the number of PUs and then contrast control flow tasks and data dependence tasks.

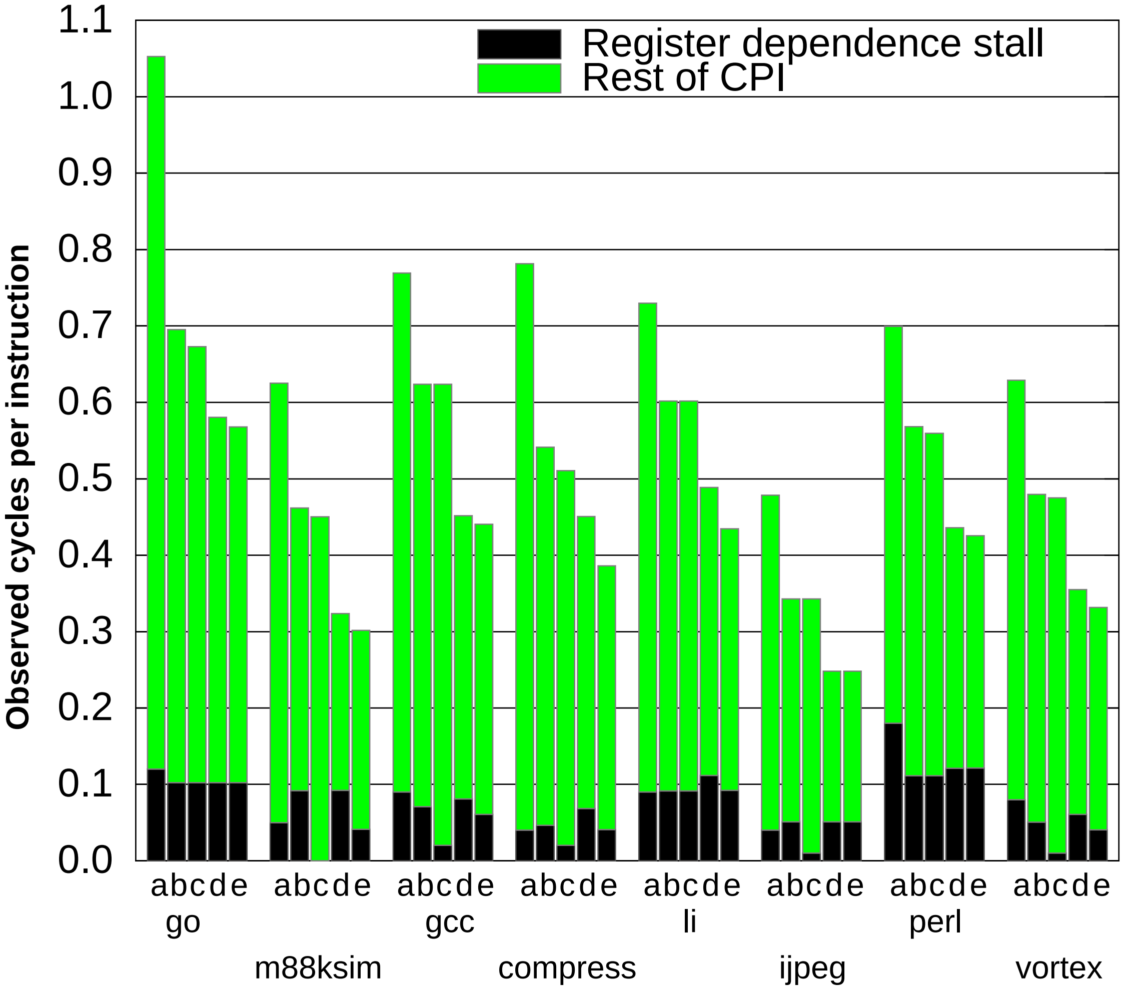


Figure 5-11: Observed register dependence stall for SPEC CINT benchmarks. All the experiments use out-of-order PUs. The five experiments marked a, b, c, d, and e are basic block tasks, control flow tasks on 4 PUs, data dependence tasks on 4 PUs, control flow tasks on 8 PUs, and data dependence tasks on 8 PUs, respectively. For compress, control flow and data dependence tasks are augmented with task size heuristics.

Increasing the number of PUs is expected to increase the number of dynamic inter-task register dependences. For example, increasing the number of PUs from 4 to 8 exposes inter-task register dependences that are 4, 5, 6, or 7 dynamic tasks apart. But Figure 5-11 and Figure 5-12 show that the average time spent by an instruction waiting for a register value in a 8 PU configuration is not much greater than that in a 4 PU configuration for

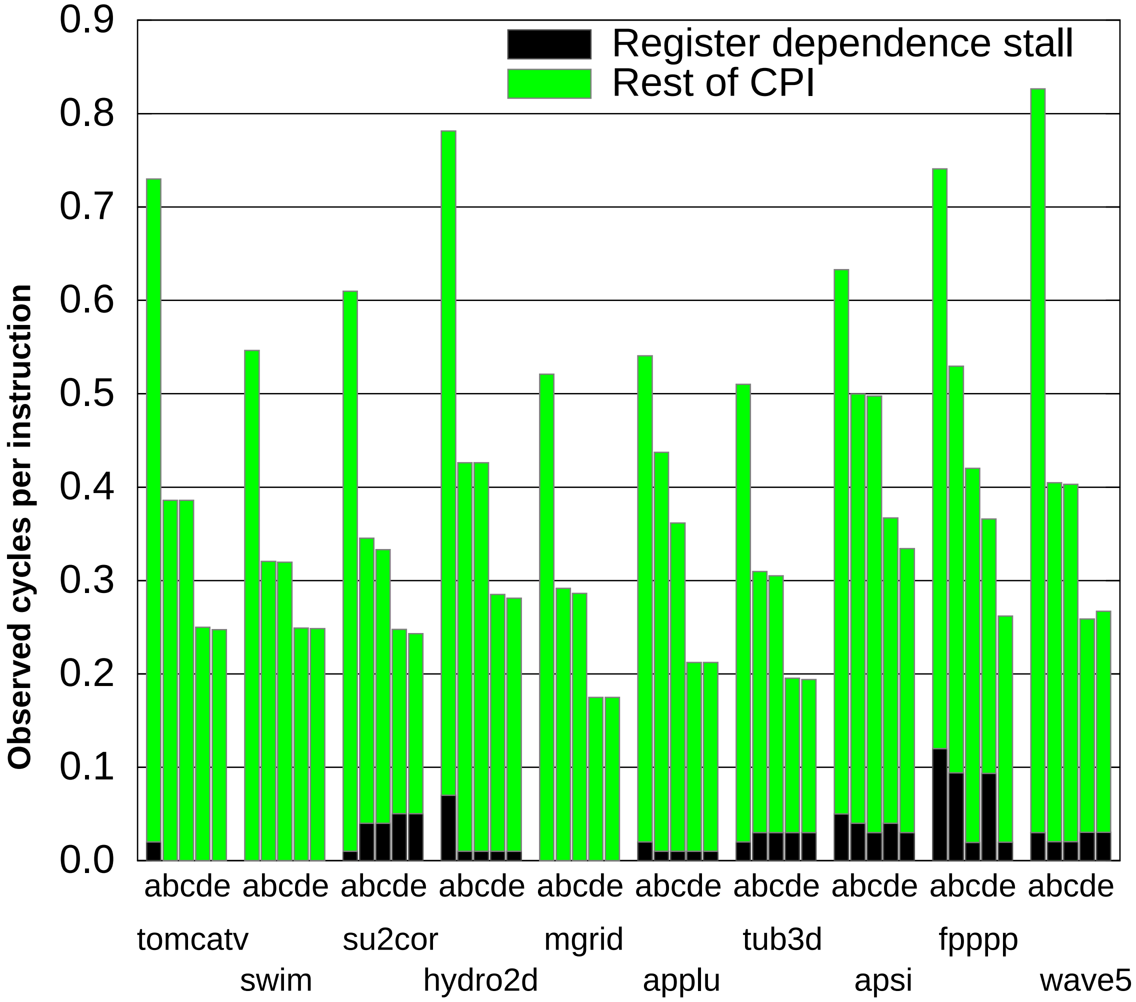


Figure 5-12: Observed register dependence stall for SPEC CFP benchmarks. All the experiments use out-of-order PUs. The five experiments marked a, b, c, d, and e are basic block tasks, control flow tasks on 4 PUs, data dependence tasks on 4 PUs, control flow tasks on 8 PUs, and data dependence tasks on 8 PUs, respectively. For fpppp, control flow and data dependence tasks are augmented with task size heuristics.

most benchmarks. The main reason for the similarity in the wait times is that most register values are short-lived and are required only in the next successor task and few are used by tasks farther in the future. Since the overall CPI decreases on increasing the number of PUs (up to some number of PUs), inter-task register dependences become more important performance factors for larger number of PUs. For control flow tasks, register wait time is

10%-24% and 17%-40% of total execution time for 4 PU and 8 PU configurations, respectively. For data dependence tasks, register wait time is 0%-24% and 12%-40% of total execution time for 4 PU and 8 PU configurations, respectively.

Data dependence tasks incur fewer wait cycles indicating that by including data dependence chains within tasks, the heuristics are able to reduce register communication delay. The reduction is significant for 128.m88ksim, 129.compress, 130.li, 132.ijpeg, 147.vortex, and 145.fpppp. The usually large reduction for 145.fpppp is because both control flow and data dependence heuristics are augmented with task size heuristics; task size heuristics split the inordinately large basic blocks of 145.fpppp into smaller blocks. Tasks generated out of the smaller blocks essentially incur dense intra-basic block register dependences.

In Figure 5-13 and Figure 5-14, I present the average number of cycles spent by an instruction waiting for memory values which are synchronized by the memory dependence prediction and synchronization hardware. Memory dependences usually span larger numb er of dynamic instructions than the size of the window established by basic block tasks (as discussed in Section 3.3.3). Consequently, basic block tasks do not incur any significant performance loss. There is no significant difference in memory wait times between control flow tasks and data dependence tasks. Although the implementation of data dependence heuristics take into account only a few memory dependences and all register dependences, data dependence tasks do not incur more memory wait time in comparison to control flow tasks. In 124.m88ksim and 145.fpppp a few memory dependences, which are not identified by simple compiler analyses, are exposed in data dependence tasks causing an increase in memory wait time.

In Figure 5-15 and Figure 5-16, I present the fraction of all register dependences as a function of the number of dynamic tasks that the dependences span. This measurement was made on data dependence tasks. For the integer benchmarks except for 132.ijpeg, almost 50% of all register dependences are included within tasks. Most inter-task register

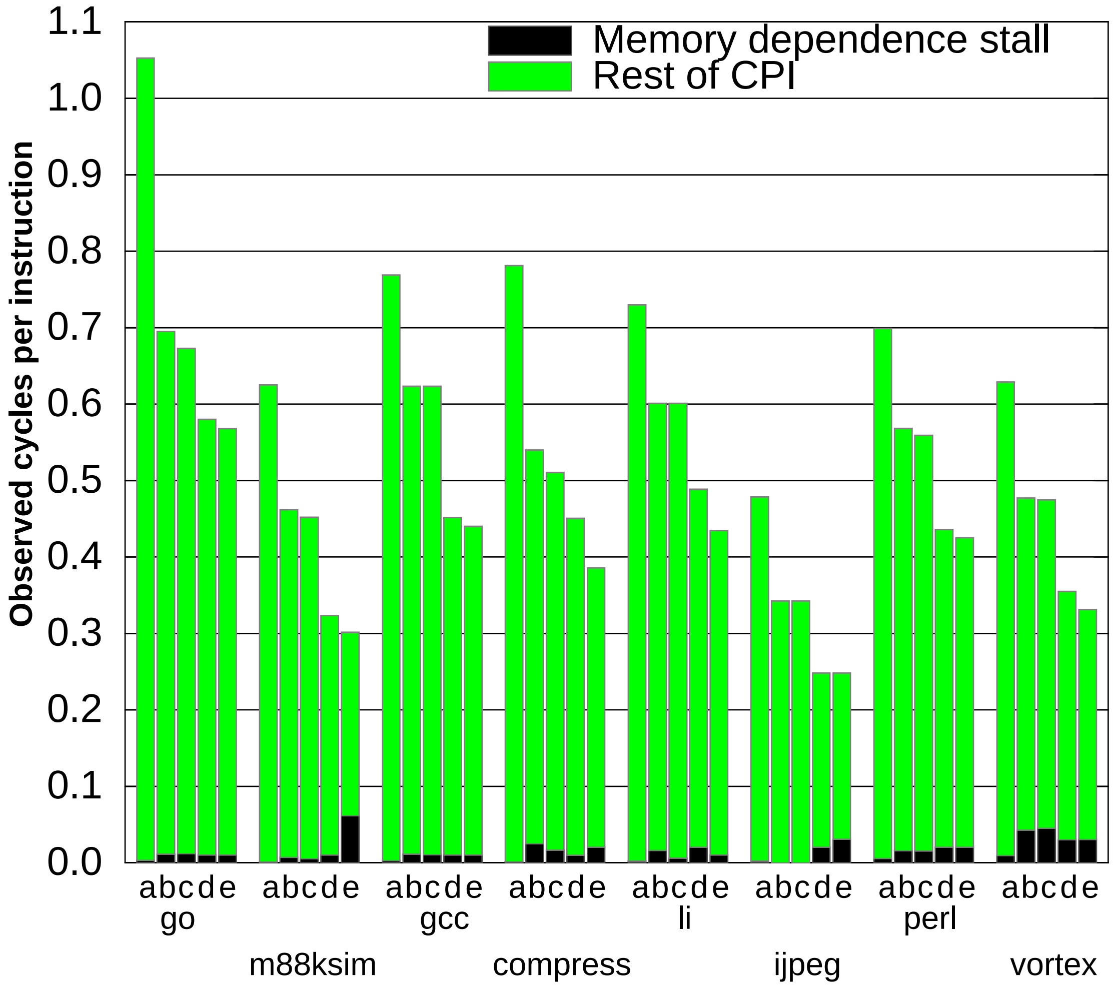


Figure 5-13: Observed memory dependence stall for SPEC CINT benchmarks. All the experiments use out-of-order PUs. The five experiments marked a, b, c, d, and e are basic block tasks, control flow tasks on 4 PUs, data dependence tasks on 4 PUs, control flow tasks on 8 PUs, and data dependence tasks on 8 PUs, respectively. For compress, control flow and data dependence tasks are augmented with task size heuristics.

dependences span between tasks that are either 1 to 3 dynamic tasks or more than 20 tasks (not shown) apart. For the floating point benchmarks and 132.ijpeg, almost 70% of all register dependences are included within tasks. Most inter-task register dependences span between adjacent tasks or tasks that are more than 20 tasks (not shown) apart. These graphs corroborates measurements previously made by Franklin and Sohi [39].

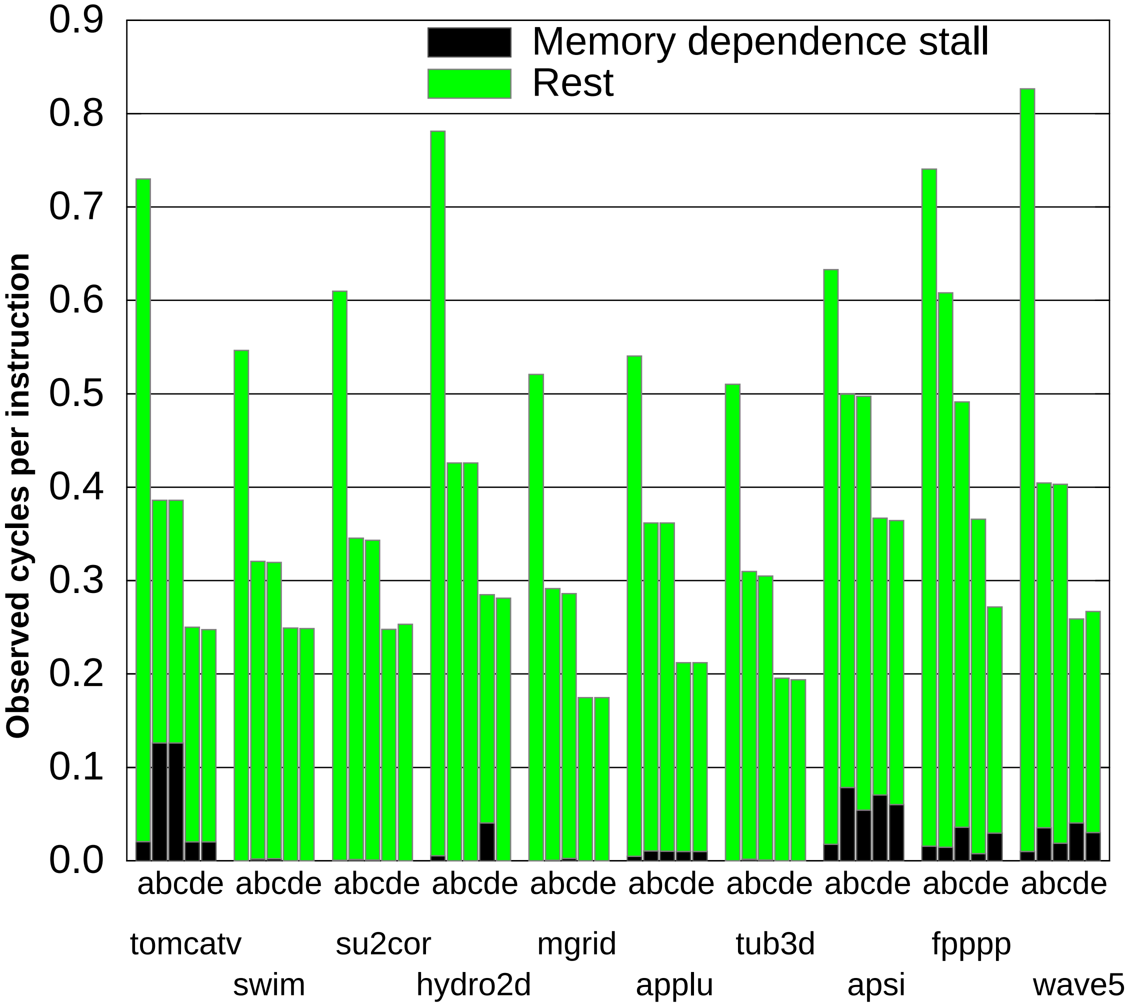


Figure 5-14: Observed memory dependence stall for CFP benchmarks. All the experiments use out-of-order PUs. The five experiments marked a, b, c, d, and e are basic block tasks, control flow tasks on 4 PUs, data dependence tasks on 4 PUs, control flow tasks on 8 PUs, and data dependence tasks on 8 PUs, respectively. For fpppp, control flow and data dependence tasks are augmented with task size heuristics.

In Figure 5-17 and Figure 5-18, I present the fraction of all memory dependences as a function of the number of dynamic tasks that the dependences span. Unlike register dependences, memory dependences span between many dynamic tasks. Inter-task memory dependencies can either cause memory dependence misspeculation or delay due to communication. While only a small fraction of register values are communicated from one

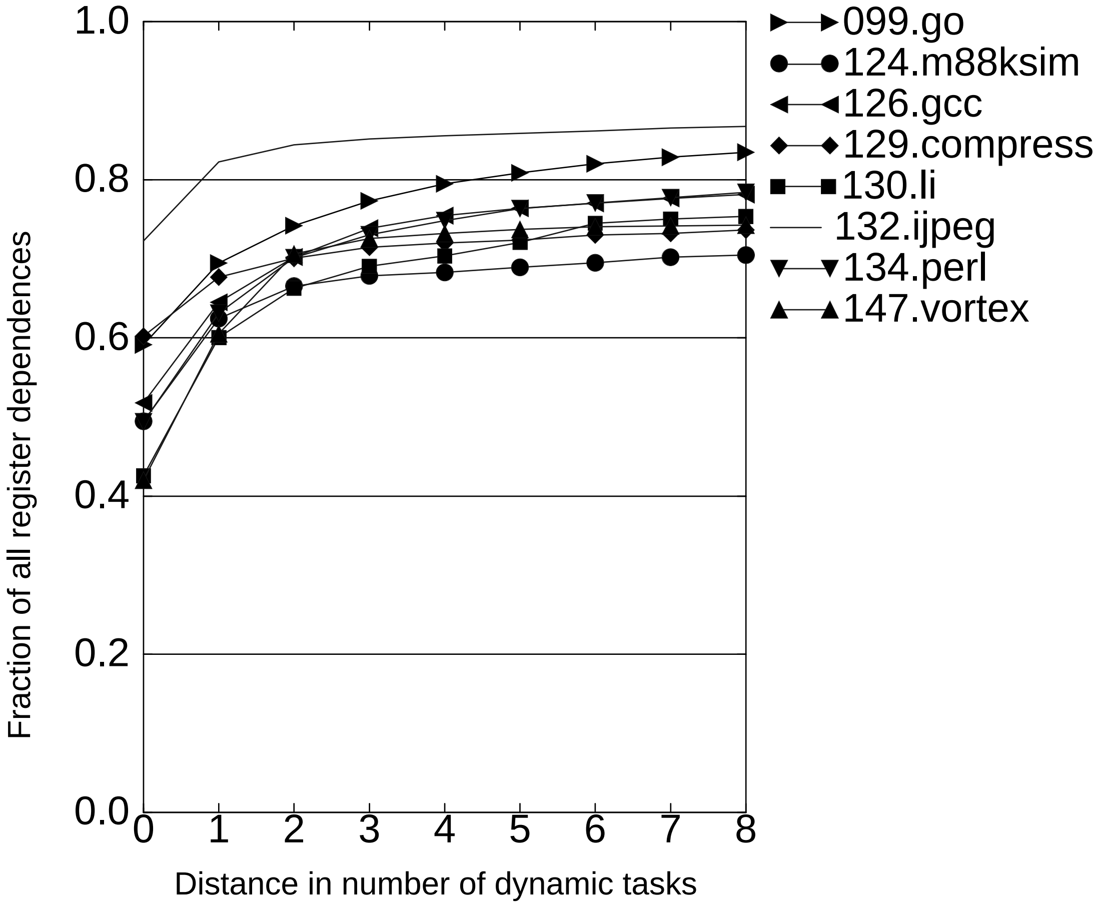


Figure 5-15: Inter-task register dependences for CINT benchmarks. The experiments use data dependence tasks.

task to another, a larger fraction of memory values are communicated. For the integer benchmarks, about 20%-40% of all memory dependences span up to 10 dynamic tasks but most of the rest of the dependences span more than 100 dynamic tasks. For floating benchmarks, there are no sharp knees in the curves indicating that memory dependences for floating point benchmarks span a wide range of number of dynamic tasks.

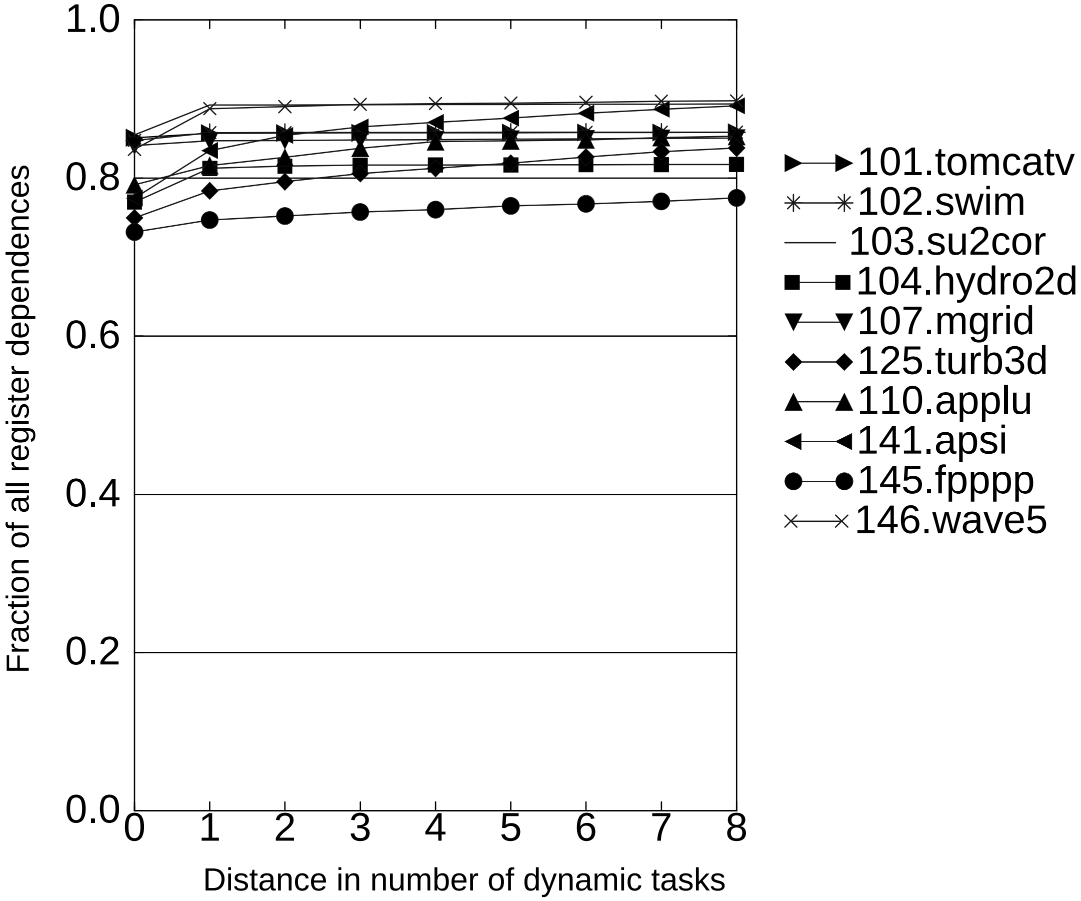


Figure 5-16: Inter-task register dependences for CFP benchmarks. The experiments use data dependence tasks.

5.3.3 Window span

To study the window established by a Multiscalar hardware configuration, I analyze the window span for the integer and the floating point benchmarks. Recall that window span is the range of all dynamic tasks in flight in the entire processor. The average size of tasks, the number of PUs, and the control flow prediction accuracy determines the size of window span. Window span is computed using the following equation where *Tasksize* is the average task size, *Pred* is the average inter-task control flow prediction accuracy, and *N* is

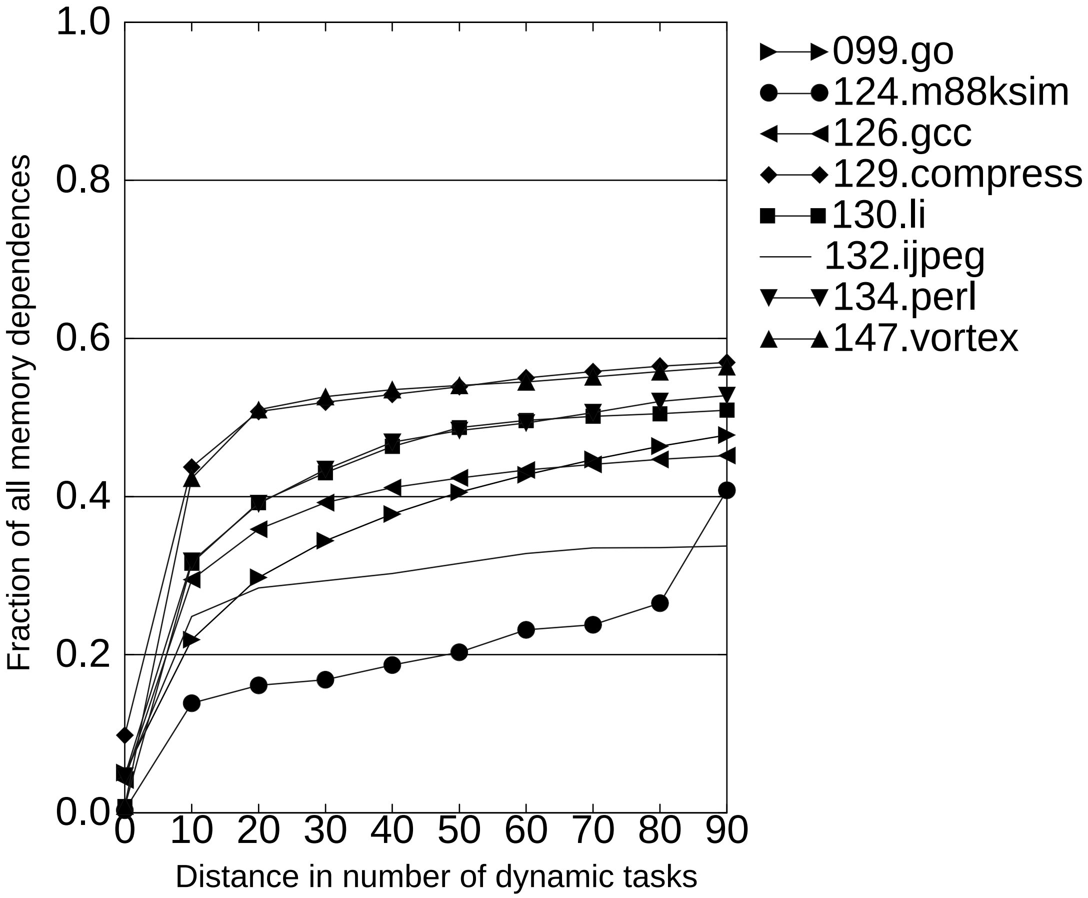


Figure 5-17: Inter-task memory dependences for CINT benchmarks. The experiments use data dependence tasks.

the number of PUs:

Although *Pred* may change slightly with increasing number of PUs, the overall effect on the window span is minimal.

In Figure 5-19 and Figure 5-20, I plot the window span measured in number of dynamic instructions as a function of the number of PUs for the integer and the floating point

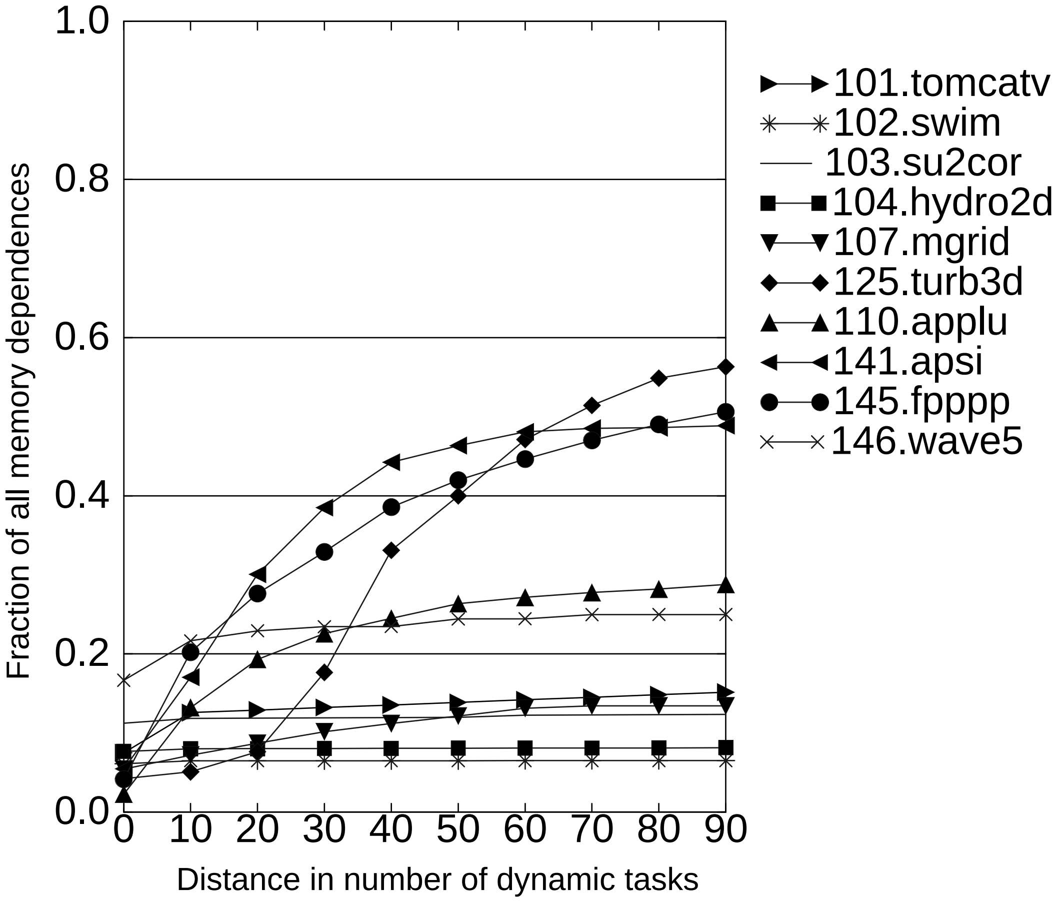


Figure 5-18: Inter-task memory dependences for CFP benchmarks. The experiments use data dependence tasks.

benchmarks, respectively, for data dependence tasks. Due to the small size of tasks, window span of most integer benchmarks are in the modest range of 25-80 instructions and 45-140 instructions for 4 and 8 PU organizations, respectively. Window spans of 134.perl and 146.vortex are increase almost linearly with the number of PUs because their prediction accuracies are high. Window span of most floating point benchmarks is considerably larger than those of their integer counterparts due to the large size of tasks and high prediction accuracy. For the floating point benchmarks, window spans are in the range of 100-400 instructions and 250-800 instructions for 4 and 8 PU configurations, respectively.

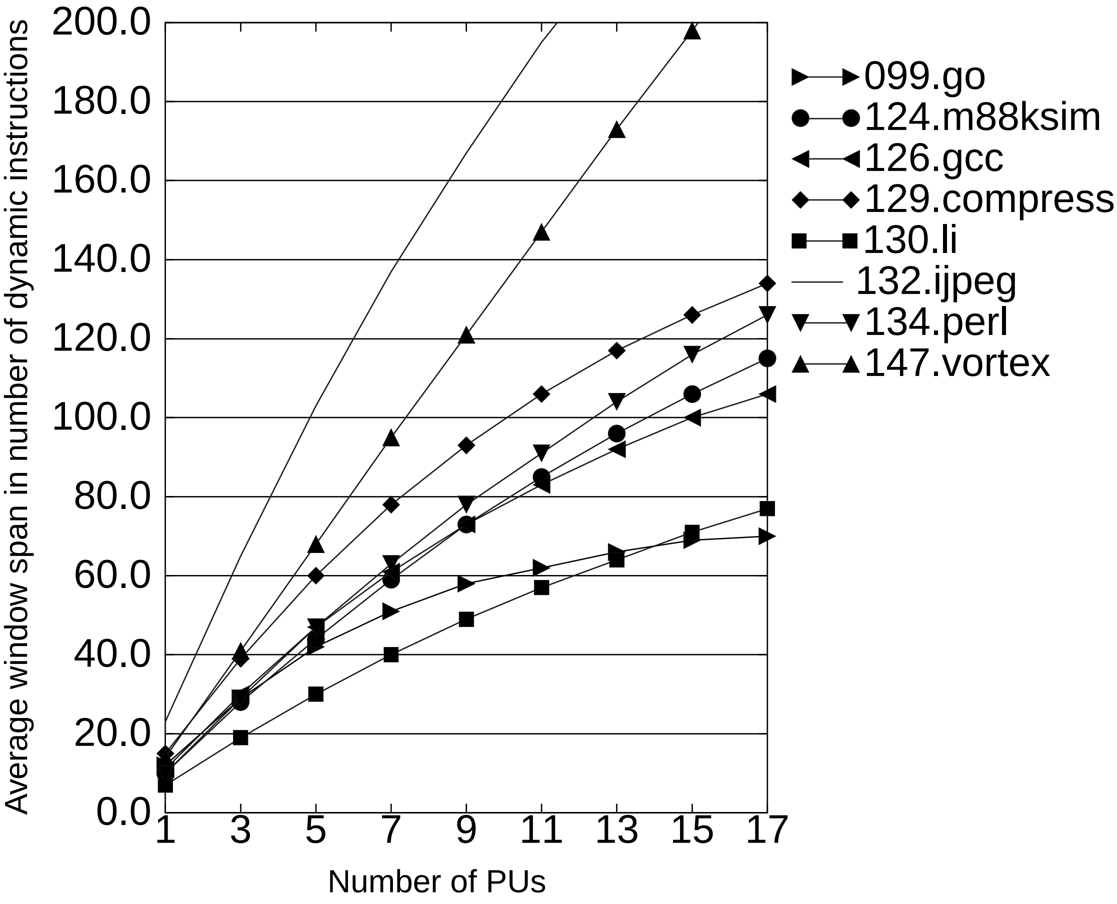


Figure 5-19: Window span of CINT benchmarks for data dependence tasks.

Figure 5-21 and Figure 5-22 show the window spans of the integer and the floating point benchmarks, respectively, for control flow tasks. Due to better inter-task control flow speculation accuracy, control flow tasks have larger spans than their data dependence task counterparts. Otherwise, the window spans for data dependence tasks and control flow tasks are similar in that the window spans of the benchmarks either flatten out or grow alike.

Figure 5-23 and Figure 5-24 show the window spans of the integer and the floating point benchmarks, respectively, for basic block tasks. Due to the significantly smaller sizes and

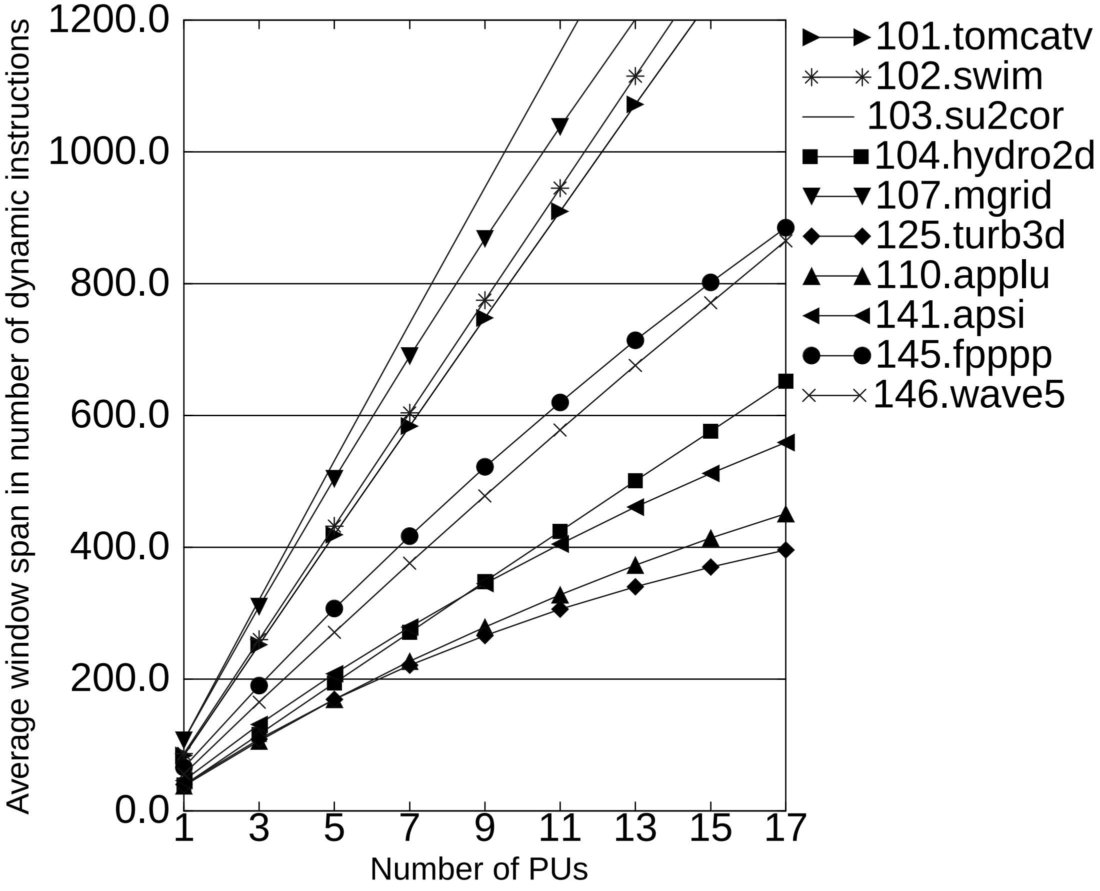


Figure 5-20: Window span of CFP benchmarks for data dependence tasks.

lower prediction accuracies of basic block tasks, the window spans are considerably smaller than those for control flow tasks and data dependence tasks. The importance of predicting at the granularity of tasks, without losing prediction accuracy, is demonstrated by this graph. The graph indicates that the amount of parallelism that is exposed through branch prediction (which is used by most modern superscalar processors) is significantly less than that exposed by task-level prediction.

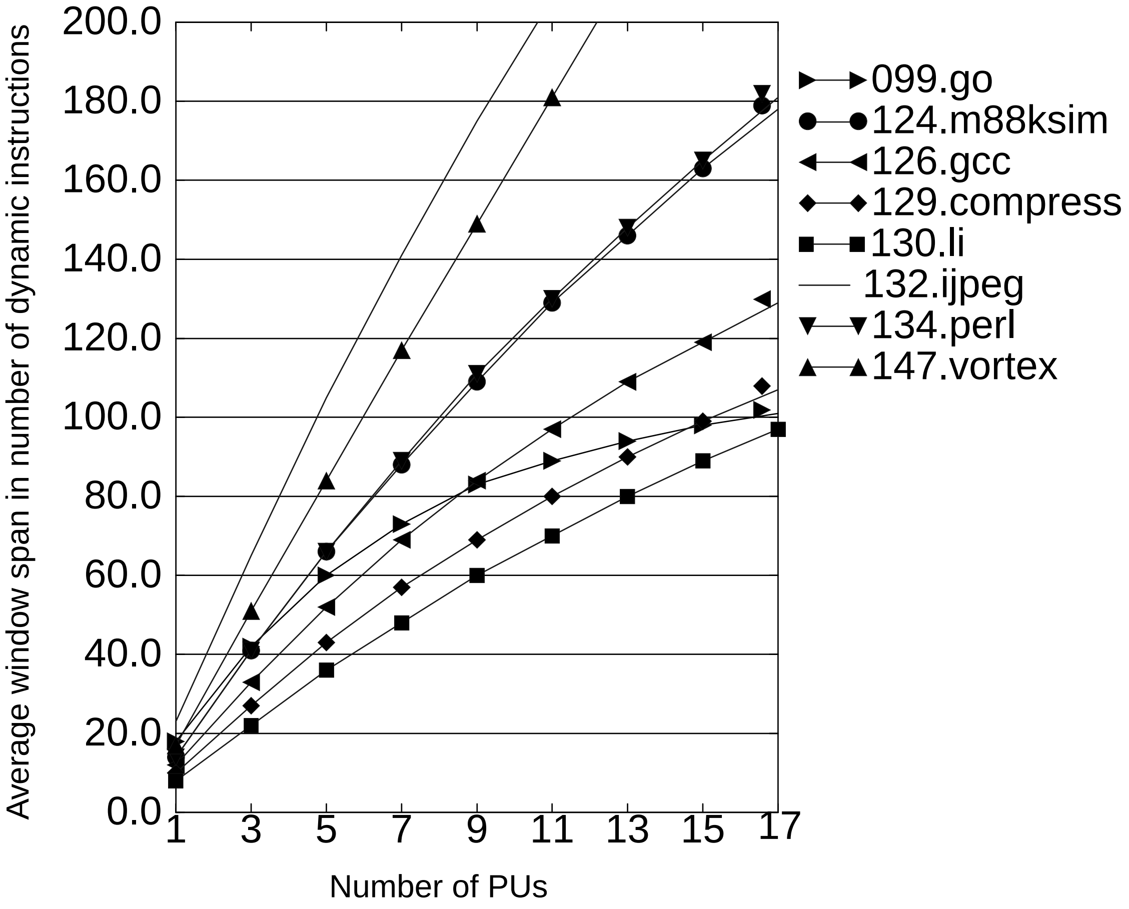


Figure 5-21: Window span of CINT benchmarks for control flow tasks.

5.3.4 Experiments related to register communication

I now present measurements of various aspects of register communication. I measure (1) the impact of the various register communication strategies, (2) the impact of register communication scheduling on overall performance, (3) the overhead of release instructions, and (4) the effectiveness of dead register analysis.

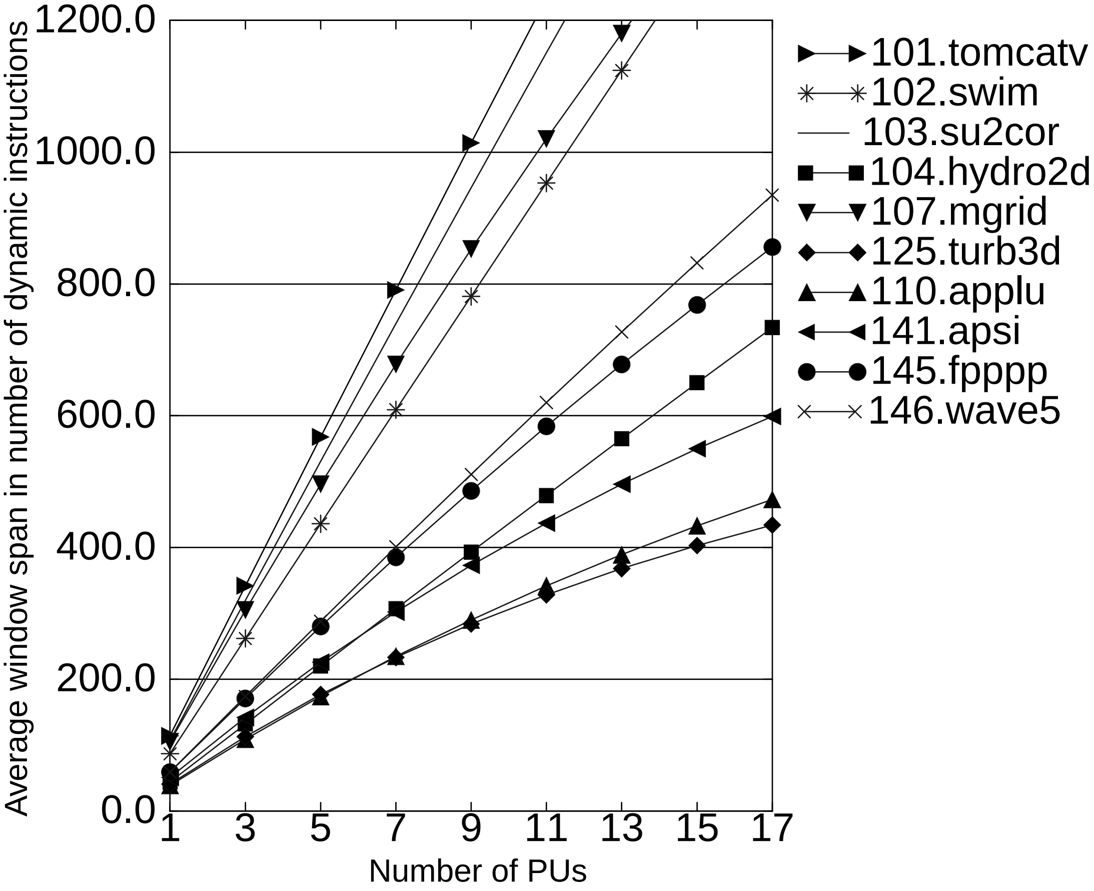


Figure 5-22: Window span of CFP benchmarks for control flow tasks.

5.3.4.1 Effectiveness of register communication strategies

The performance achieved by the register communication strategies depends on the effectiveness of each scheme in avoiding stalling consumers by sending register values early. Figure 5-25 and Figure 5-26 show the overall performance achieved by the integer and floating point benchmarks using the various strategies. The performance improvement achieved by the most sophisticated strategy, last_send, is significant for all the benchmarks. For the integer benchmarks other than 129.compress, eager_send strategy improves performance 12%-39%. 129.compress squashes excessively under eager_send

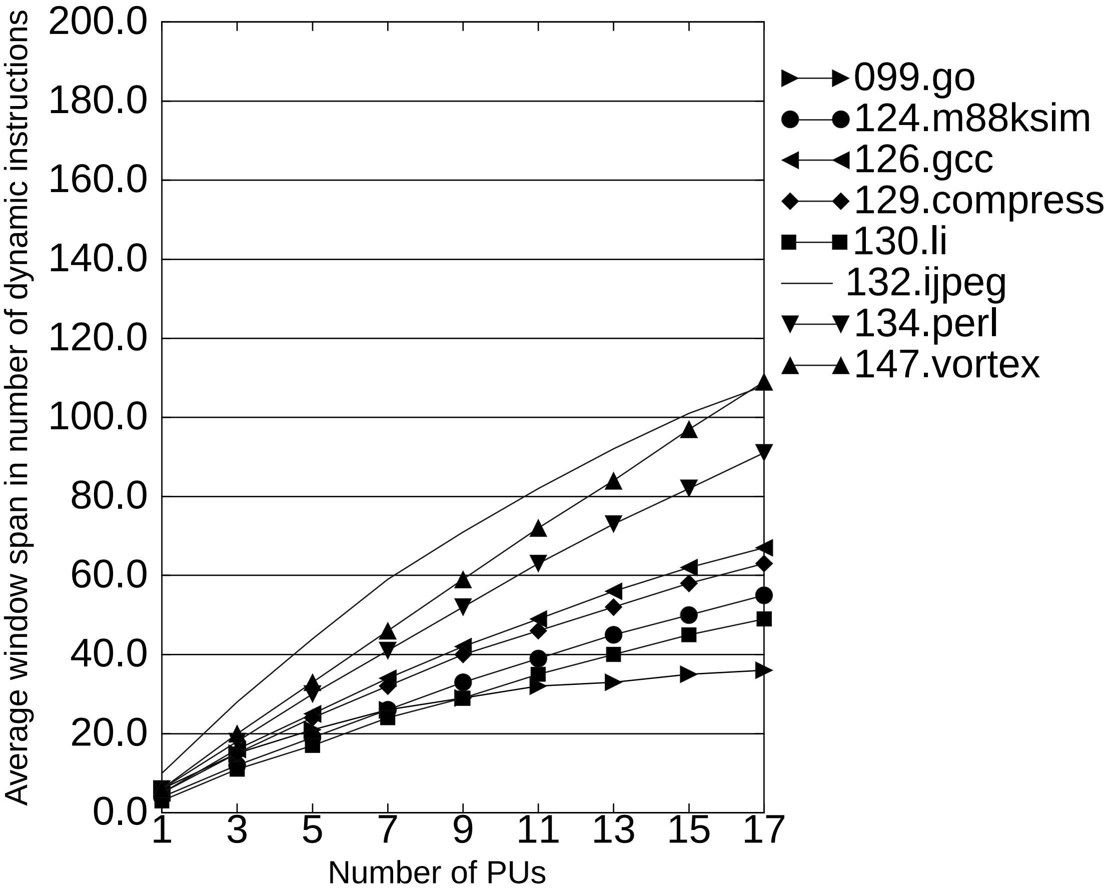


Figure 5-23: Window span of CINT benchmarks for basic block tasks.

strategy offsetting any performance gain achieved by sending register values early. Last_send strategy improves performance for the benchmarks with large tasks[4]. For small tasks it is likely that any modification of a register is the last modification, so that eager_send strategy marks register values to be sent early without causing any squashes. 126.gcc and 130.li demonstrate this behavior, whereas all the other benchmarks improve 6%-22% under last_send strategy when compared with eager_send strategy (Figure 5-25). Spec_send performs worse than last_send for most cases, except 130.li, because the

[4] If the squash model used by eager_send strategy selectively squashes only dependent instructions, then eager_send strategy may perform better than last _end strategy.

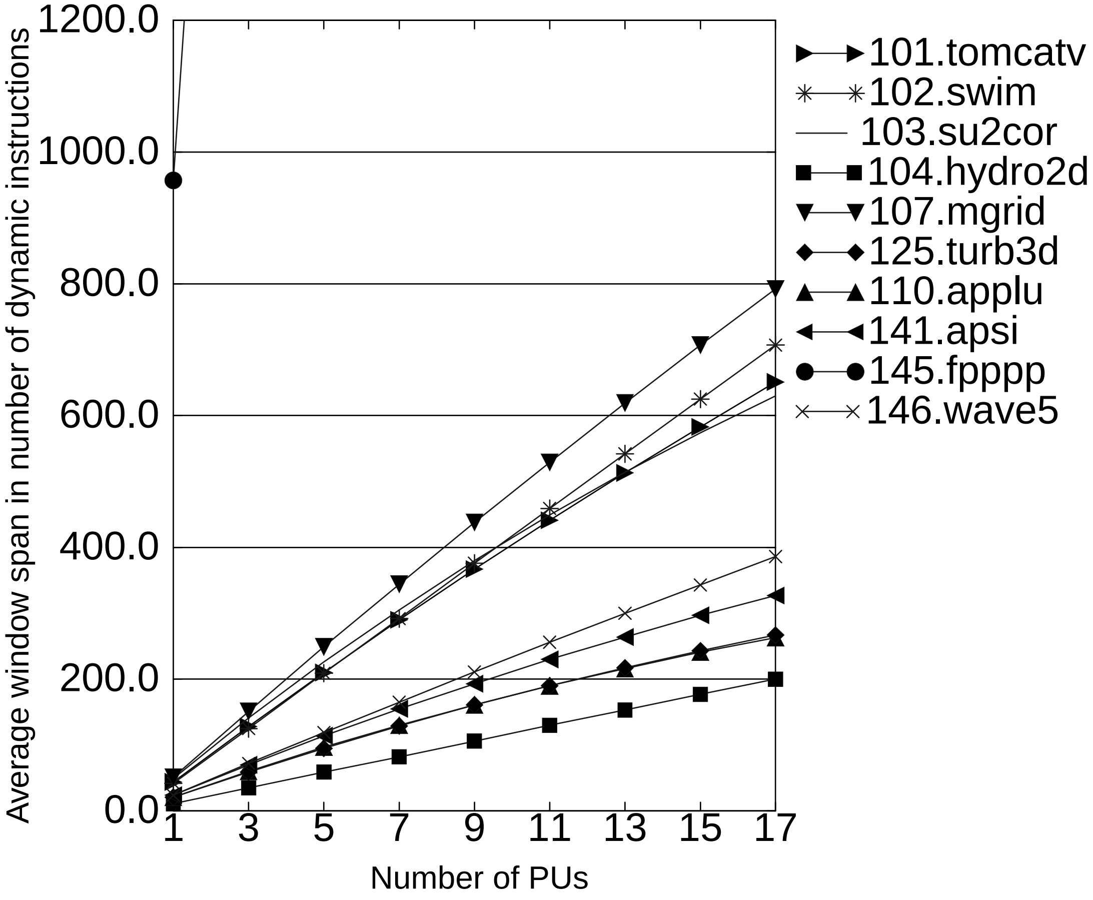


Figure 5-24: Window span of CFP benchmarks for basic block tasks.

squashes caused by it, even though infrequent, are expensive. Spec-send essentially exposes the internal branches of tasks incurring performance loss. Since the floating point benchmarks have large tasks, last_send strategy is even more effective. Eager_send strategy improves performance 5%-50% and last_send strategy adds 9%-42% over eager_send strategy (Figure 5-26). Again, spec_send performs worse because the squashes caused by it discard many instructions because floating point tasks are usually large in size. Selectively squashing only those instructions that are dependent on misspeculated register values may improve spec_send performance.

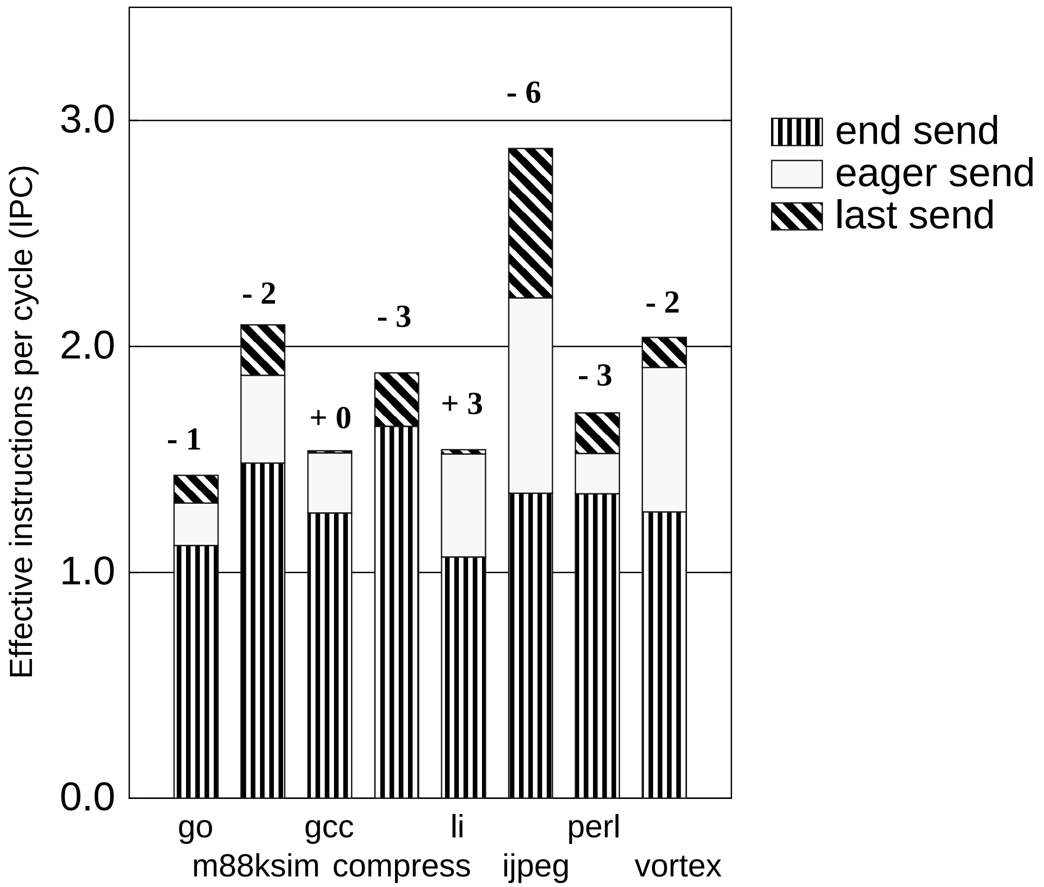


Figure 5-25: Register communication strategies for CINT benchmarks. The experiment uses data dependence tasks executing on 4 out-of-order PUs. The numbers on above the bars show the percentage improvement of spec_send over last_send.

5.3.5 Impact of register communication scheduling

In addition to the register communication strategies evaluated above, scheduling moves computation so that producers are executed early and consumers are executed late, as well as restructures loops to accelerate communication of induction variable values. Figure 5-27 and Figure 5-28 show the overall performance improvements achieved by scheduling register communication, via code motion and loop restructuring, for the integer and floating point benchmarks using data dependence tasks executing on out-of-order

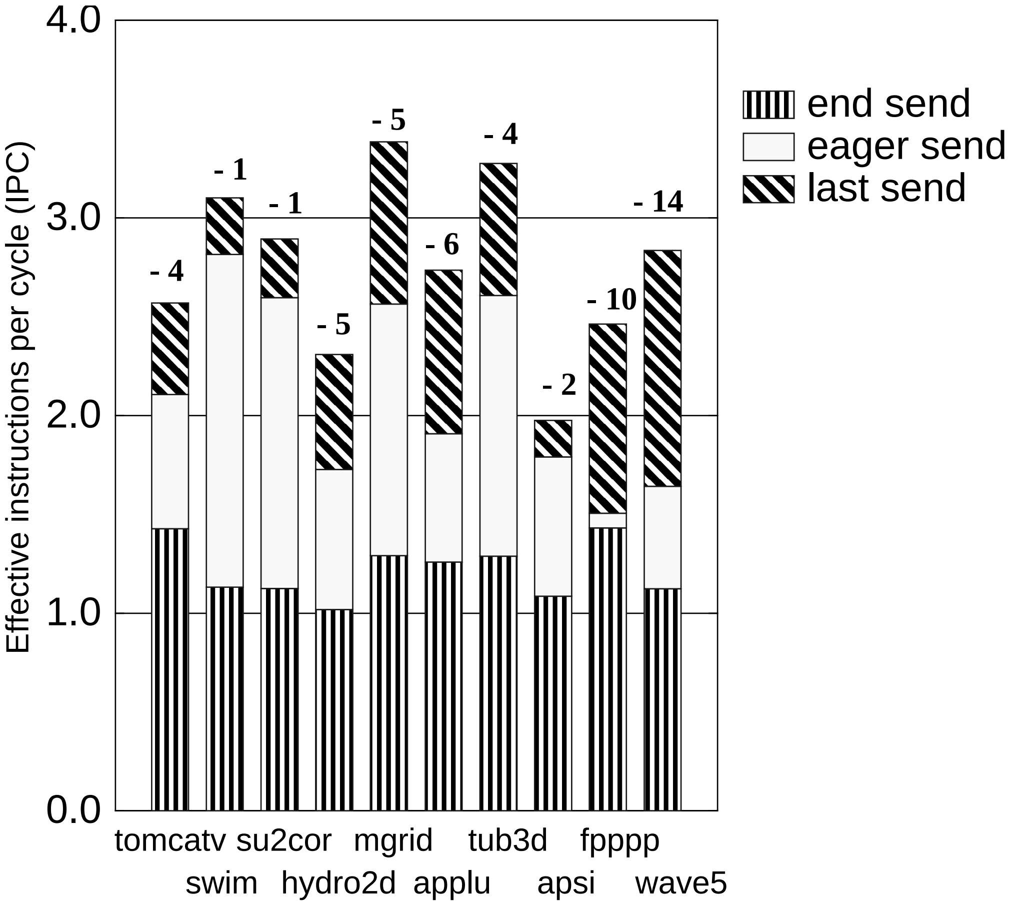


Figure 5-26: Register communication strategies for CFP benchmarks. The experiment uses data dependence tasks executing on 4 out-of-order PUs. The numbers on above the bars show the percentage improvement of spec_send over last_send.

PUs, respectively. Using out-of-order PUs, the floating point programs respond more to compiler scheduling than the integer programs. In the context of out-of-order PUs, compiler scheduling impacts performance in two ways: (1) Instructions that are crucial to communication are moved so that they are buffered in the reorder buffer in favorable order with respect to the other instructions (i.e., early for producers and late for consumers) and (2) the crucial instructions are prioritized favorably to utilize the resources of the PU by this ordering (e.g., cache ports).

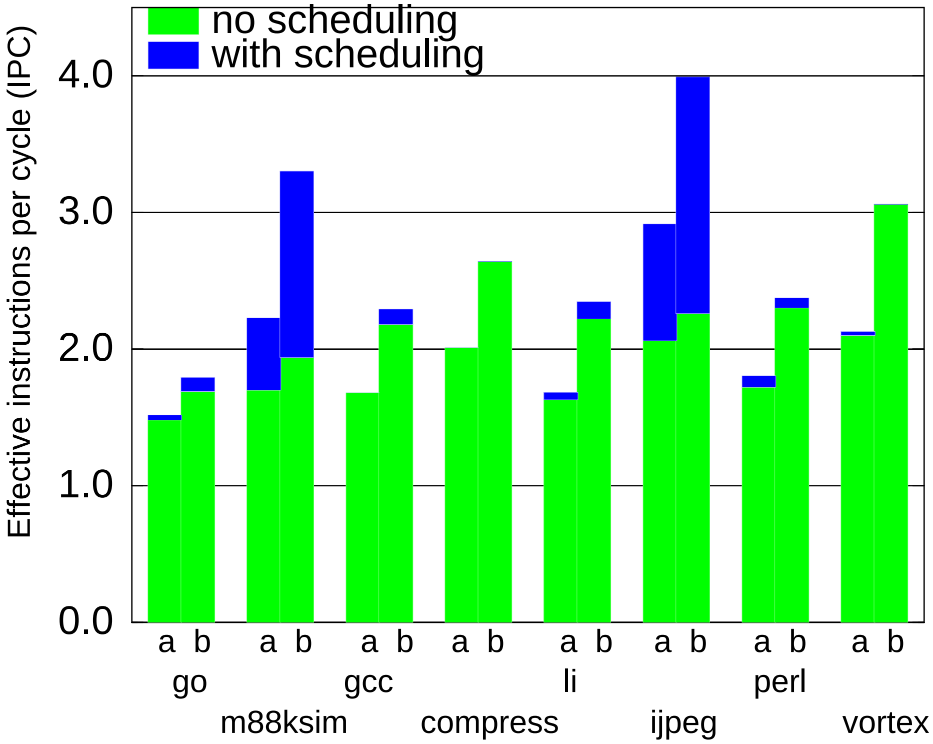


Figure 5-27: Register communication scheduling for CINT benchmarks. The two experiments marked a and b use data dependence tasks executing on 4 and 8 out-of-order PUs, respectively. The experiments include both code motion and loop restructuring.

Even for out-of-order PUs, which are capable of hiding latency, compiler scheduling improves performance significantly for 124.m88ksim and 132.ijpeg. The integer benchmarks improve <1%-30% and <1%-45% for 4 and 8 PUs, respectively (Figure 5-27). 099.go, 126.gcc, 130.li, and 134.perl show modest improvements, where as 129.compress and 147.vortex respond less to scheduling. 129.compress has a tight recurrence in its most frequent loop which cannot broken by scheduling and other dependences are rescheduled dynamically. 147.vortex has unusually high branch prediction accuracy (> 99%) and out-of-order PUs establish an almost accurate window from which they reschedule instructions almost perfectly. There are several reasons for the modest performance improvements of the integer benchmarks due to scheduling. Unlike loop restructuring, code

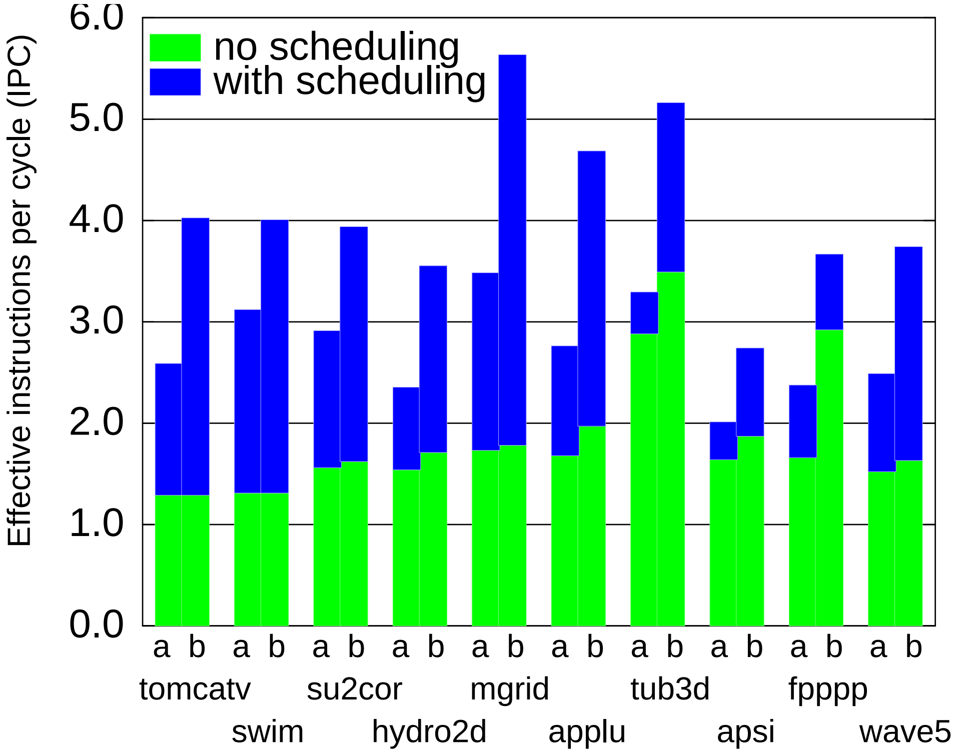


Figure 5-28: Register communication scheduling for CFP benchmarks. The two experiments marked a and b use data dependence tasks executing on 4 and 8 out-of-order PUs, respectively. The experiments include both code motion and loop restructuring.

motion is hindered by intra-task control flow dependences and intra-task ambiguous data dependences. Also, since data dependence tasks already optimize for inter-task data dependences, scheduling has fewer opportunities to optimize further. For the benchmarks that respond to scheduling, the improvements grow in both in absolute magnitude and as a percentage over the base case on increasing the number of PUs from 4 to 8.

The floating point benchmarks improve 12%-58% and 20%-69% for 4 and 8 PUs, respectively (Figure 5-28). The floating point benchmarks improve significantly mainly because loop induction variables are moved to the top of the loops and other computation dependent on the induction variables also get accelerated. Out-of-order PUs do not have large enough reorder buffers to capture entire loop bodies, which are large, and accelerate induction variables dynamically.

Figure 5-29 and Figure 5-30 show the overall performance improvements achieved by scheduling register communication for the integer and floating point benchmarks using data dependence tasks executing on in-order PUs, respectively. The trends in performance improvement observed for out-of-order PUs with respect to the number of PUs hold true for in-order PUs as well. The integer benchmarks improve <1%-33% and <1%-47% for 4

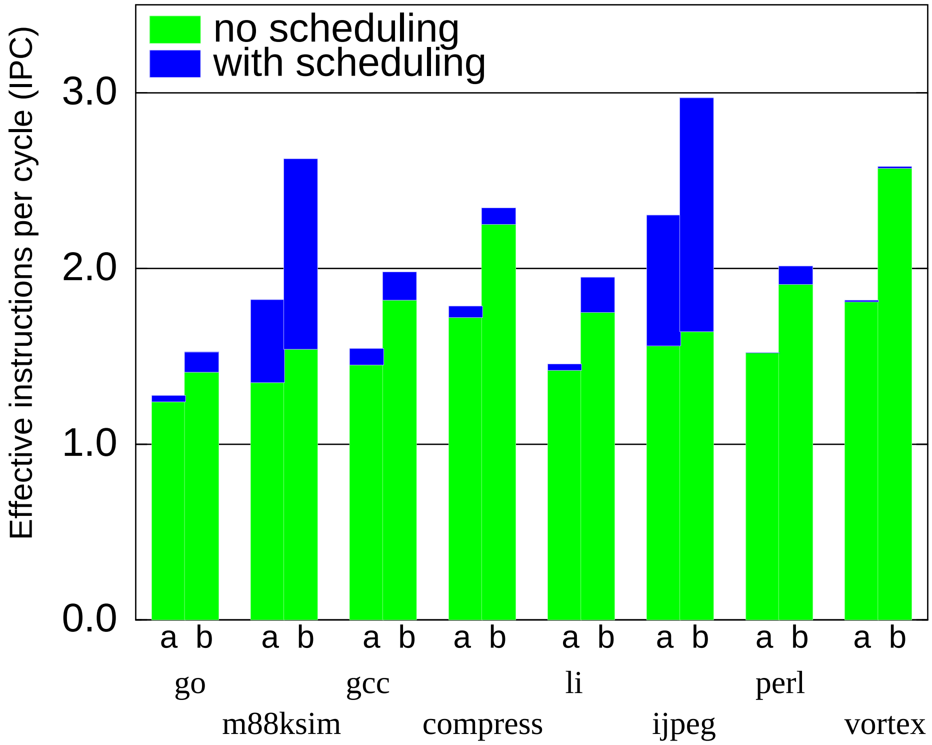


Figure 5-29: Register communication scheduling for CINT benchmarks. The two experiments marked a and b use data dependence tasks executing on 4 and 8 in-order PUs, respectively. The experiments include both code motion and loop restructuring.

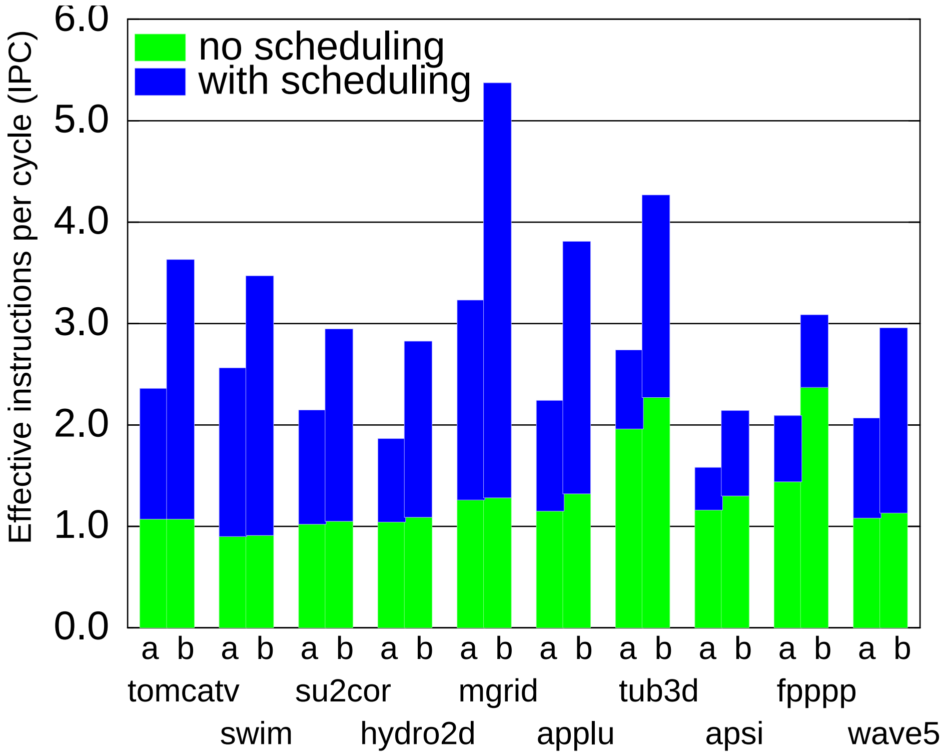


Figure 5-30: Register communication scheduling for CFP benchmarks. The two experiments marked a and b use data dependence tasks executing on 4 and 8 in-order PUs, respectively. The experiments include both code motion and loop restructuring.

and 8 PUs, respectively. The floating point benchmarks improve 27%-65% and 39%-77% for 4 and 8 PUs, respectively. Compared to out-of-order PUs, in-order PUs achieve performance improvement a little more in absolute magnitude and as a percentage of the base case. The similarity in improvements between in-order and out-of-order PU configurations indicate that register communication delays are not hidden by out-of-order PUs with modest-sized reorder buffers and ready lists.

5.3.5.1 Overhead of release instructions

In Table 5-8, I present the number of dynamic release instructions executed as a fraction of all dynamic instructions. These experiments include dead register optimization. The

Table 5-8: Overhead of dynamic release instructions for SPEC95 benchmarks.

Benchmarks	fraction
099.go	2.0%
124.m88ksim	2.0%
126.gcc	2.0%
129.compress	1.5%
130.li	3.7%
132.ijpeg	1.0%
134.perl	3.3%
147.vortex	1.8%

Benchmarks	fraction
101.tomcatv	0.0%
102.swim	0.0%
103.su2cor	0.0%
104.hydro2d	0.0%
107.mgrid	0.0%
110.applu	0.2%
125.turb3d	0.0%
141.apsi	0.1%
145.fpppp	0.1%
145.wave5	0.0%

overhead due to release instructions depends on the size of the tasks. Larger tasks encapsulate more register live ranges within them, reducing the amount of communication performed in the program. Overall, avoiding extra instructions by annotating existing instructions to convey register communication information and not sending dead registers are effective in keeping the overhead small. For the integer benchmarks, the overhead is of the order of 2%-3% and for the floating point benchmarks, there is little overhead.

5.3.5.2 Effectiveness of dead register analysis

In Table 5-9, I present the reduction in register communication traffic by employing dead register analysis. In this experiment, register communication traffic is measured as the number of hops a register value makes on the ring per dynamic instruction of the program. The reason for using this metric is that dead register optimization reduces traffic in two ways: (1) By not propagating dead values and (2) by stopping dead values from making unnecessary hops on the register communication ring. Dead register optimization is aimed at reducing the number of register values sent on the ring and the number of hops each register value makes on the ring. This metric is independent of the IPC achieved, so

Table 5-9: Impact of dead register optimization on SPEC95 benchmarks. The experiments use 4 PUs. The column titled base shows the base case as the average number of hops a register value makes on the ring per dynamic instruction. The column titled decrease shows the percent reduction in the traffic due to dead register analysis.

Benchmarks	base	decrease	Benchmarks	base	decrease
099.go	1.00	33.9%	101.tomcatv	0.22	75.7%
124.m88ksim	0.77	25.6%	102.swim	0.22	86.2%
126.gcc	0.94	26.9%	103.su2cor	0.16	58.2%
129.compress	0.92	5.7%	104.hydro2d	0.32	74.0%
130.li	0.46	19.9%	107.mgrid	0.30	60.8%
132.ijpeg	0.47	42.6%	110.applu	0.41	67.3%
134.perl	0.80	22.0%	125.turb3d	0.55	66.8%
147.vortex	0.77	20.0%	141.apsi	0.55	58.9%
			145.fpppp	0.62	74.3%
			145.wave5	0.31	63.5%

that the effectiveness of the compiler, in terms of the number of dead register values stopped from occupying the ring unnecessarily, is isolated without confusing with timing issues. The compiler is effective in reducing the traffic for both the integer and floating point benchmarks. For the integer benchmarks, the compiler analysis reduces the traffic by 5%-42% and for the floating point benchmarks, the reduction is in the range of 58%-75%. Larger tasks lead to more reduction in register communication traffic under dead register optimization because larger tasks include more register live ranges resulting in more dead registers at the exit of the tasks. The reduction indicates that many register values are produced and consumed within the same task or a few dynamic tasks apart. The reduction is the ratio between the number of register values local to a task and the number of all register values created, as discovered by the compiler[5]. In the floating point benchmarks, mostly only induction variables remain live across tasks, resulting in substantial reduction in register communication traffic.

[5] Due to control flow, the compiler analysis has to be conservative. There may be registers that are dead in one control flow path and not another; the compiler cannot mark those registers as dead.

Chapter 6

Conclusions

Straightforward approaches of scaling up the centralized hardware structures of superscalar processors to extract more parallelism may hinder the speed at which the processors may be clocked, limiting overall performance. By employing a novel, distributed hardware organization, the Multiscalar architecture proposes to extract more parallelism at high clock speeds. Multiscalar provides hardware support for control flow speculation, register value communication and memory dependence speculation to achieve high performance on sequential applications.

In this thesis, I studied the fundamental interactions between sequential programs and the novel features of the Multiscalar architecture, from the standpoint of performance. I determined and explored the key implications of the interactions for the compiler. I devised, implemented, analyzed and experimented with compiler techniques to improve performance. Since this thesis is the first attempt at investigating this problem, I identified the key issues involved in the problem of compiling for the Multiscalar architecture and explored a few compiler optimization opportunities instead of proposing the best technique to solve a specific optimization problem. To make the investigation concrete, I constructed a compiler and evaluated the impact of compiler techniques on performance of the SPEC95 [98] benchmark suite.

6.1 Thesis summary

The compiler partitions sequential programs into code fragments called tasks, which are not necessarily independent, to map the programs on distributed processing units for execution. To maintain sequential program semantics, the responsibility of honoring inter-task register dependences, inter-task control flow dependences, and inter-task memory dependences is shared between the hardware and the compiler. In the Multiscalar architecture implementation that I investigated, the compiler partitions sequential programs into tasks. Inter-task register dependences are specified by the compiler, inter-task control flow dependences are specified by the compiler and inter-task memory dependences are handled entirely by the ARB.

Task selection crucially affects overall performance achieved by the Multiscalar architecture. Control flow speculation, register communication, memory dependence speculation, load imbalance, and task overheads are fundamental performance issues. Task size, inter-task control flow dependences and inter-task data dependences are important characteristics of tasks that impact these performance issues.

Important criteria to consider during task selection are: (1) Tasks should be neither small not large; a small task may incur overheads that may not be amortized over the execution of the task and high register communication delays, where as a large task may incur memory dependence misspeculations and ARB overflows. (2) The number of successors of a task should be as many as can be tracked by the control flow speculation hardware; reconvergent control flow paths can be exploited to generate tasks which include multiple basic blocks without taxing the prediction hardware and avoid exposing hard-to-predict branches to the prediction hardware. (3) Data dependences should be included within tasks to avoid communication and synchronization delays or misspeculation and roll back penalties. If a data dependence cannot be included within a task, then the dependence should be exposed such that the producer and consumer instructions involved in the dependence are scheduled favorably (i.e., the producer is executed early and the consumer is executed late in their respective tasks).

Based on these criteria, I devised and implemented the following heuristics to perform task selection: (1) Terminate tasks at entry and exit of loops and function invocations to avoid large tasks. (2) Include multiple basic blocks to avoid small tasks without exceeding the number of successors that may be tracked by the hardware. (3) Expose loop back edges instead of branches internal to the loop to improve control flow speculation accuracy and capture loop-level parallelism. (4) Include register and simple memory dependences that span multiple basic blocks within tasks. (5) If a data dependence is exposed then generate a task with the producer of the dependence as the root of its task.

Overlapping register communication with computation is of paramount importance to alleviate performance loss. I have developed several compiler strategies to study the effect of register communication delay on overall performance. These strategies vary the degree of overlap of communication with computation by varying the aggressiveness of the analyses. End_send strategy conservatively posts forwards of register values at task end, eager_send strategy posts speculative forwards of register values as soon as a register is modified, even though it may not be the last modification, last_send strategy posts forwards of register values only after the last modification, and spec_send strategy posts speculative forwards of register values after a register is modified, if it is likely to be the last modification.

All of these strategies vary the timing of communication but do not move the computation that generates the values involved in the communication. Extending this analysis further, I have devised and implemented a static scheduler that moves computation to hide inter-task register communication delay further. The scheduler identifies inter-task register dependences and moves the producer instruction up and the consumer instruction down in their respective tasks. The scheduler uses traditional code motion transformations and a profiling-based cost model that computes the delay incurred by an inter-task register dependence by estimating the number of cycles after the start of their respective tasks the producer and the consumer instructions execute.

I implemented the task selection heuristics, register communication strategies and scheduling, and dead register optimization in Gcc (version 2.7.2) and experimented with the SPEC95 benchmark suite using a simulator for the Multiscalar architecture. I summarize the results of my experiments and draw some conclusions.

6.2 Thesis conclusions

The important results of this thesis are: (1) The key issues in achieving performance are increasing the size of tasks to establish a large window, improving inter-task control flow speculation accuracy, reducing inter-task data dependence delays, and decreasing load imbalance. (2) With respect to basic block tasks, the task selection heuristics successfully increase the size of tasks, improve control flow speculation accuracy (normalized to the number of branches) and reduce inter-task data (mostly register) dependence delays. (3) Register communication strategies demonstrate the importance of overlapping communication with computation by sending register values as soon as they are generated. (4) Register communication scheduling, through code motion and loop restructuring, further decreases inter-task register dependence delays, by moving producer instructions up and consumer instructions down and restructuring loops so that loop induction variables are incremented at the top of loop bodies. (5) Dead register optimization significantly reduces register communication traffic by taking advantage of the fact that most register values created in a task either die within the task or within a few successor tasks.

The heuristics increase task sizes significantly with respect to basic blocks, enabling a larger window span from which to extract parallelism. The integer benchmarks have modest (tens of instructions) to large (hundreds of instructions) window spans and the floating point benchmarks have large window spans. The window spans for heuristic tasks are considerably larger than those for basic block tasks, indicating that the Multiscalar architecture has the potential to extract significantly more ILP through task-level speculation than superscalar architectures, which rely on branch prediction. In spite of including multiple basic blocks, the heuristics are effective in improving the accuracy of inter-task control flow speculation, by controlling the number of successors of tasks. By including data (mostly register) dependences within tasks, whenever possible, and by exposing the rest of data dependences so that producers and consumers are placed early and late in their respective tasks, the heuristics successfully reduce inter-task register dependence delays.

The task selection heuristics are effective in partitioning sequential programs into suitable tasks. The performance improvements achieved by the heuristics range a wide spectrum for the integer and floating point benchmarks. The heuristics extract modest to high amount of parallelism from the integer benchmarks. The heuristics are uniformly more successful in exploiting parallelism in the floating point benchmarks. For the integer benchmarks, the heuristics demarcate tasks containing a few basic blocks, which have no particular control flow structure (e.g., loop body). But for the floating point benchmarks, tasks containing entire loop bodies are predominant. Increasing the number of PUs increases the improvements for heuristic tasks, indicating that the heuristics better utilize extra hardware.

Performance improvement for control flow tasks over basic block tasks is more than that for data dependence tasks over control flow tasks. By including adjacent basic blocks within tasks, control flow heuristics include data dependences within tasks, leaving little for data dependence heuristics to optimize. In-order PU configurations show more improvement for data dependence tasks than out-of-order PUs. Using heuristic tasks, 8 in-order PUs consistently perform better than 4 out-of-order PUs, indicating that simpler, more number of PUs may perform better than fewer, more complicated PUs, if sophisticated compiler techniques are used[1].

In spite of including multiple basic blocks, control flow tasks and data dependence tasks incur fewer control flow misspeculations (counted per branch) than basic block tasks, indicating that the heuristics expose more predictable control flow to the hardware. The synergy between the heuristics and the prediction hardware is effective in improving the accuracy of control flow speculation.

The data dependence heuristic alleviates inter-task register communication delays significantly for both in-order and out-of-order PUs. Inter-task register communication delay is a more important performance factor for larger number of PUs. The magnitude of the delay does not change with the number of PUs, indicating that next-neighbor communication is the most prevalent pattern of communication.

Task overheads decrease significantly due to the larger sizes of heuristic tasks. Task overheads are less significant as the number of PUs increases. Load imbalance is a more significant performance factor for larger number of PUs. The number of cycles lost to load imbalance does not change with the number of PUs, indicating that load imbalance between dynamically adjacent tasks is more important than for tasks that are dynamically farther apart.

Performance difference effected by the register communication strategies emphasize the importance of overlapping register communication with computation. Sophisticated register communication strategy, last_send, out performs simplistic register communication strategies, end_send and eager_send, by sending register values as soon as they are created. Even for out-of-order PUs, last_send out performs end_send and eager_send, indicating that aggressive, latency-tolerant hardware cannot overcome the delay imposed by

[1] The 8 PU configuration had more peak memory subsystem bandwidth.

inter-task register communication. Brute force hardware techniques (similar to eager_send) may need more complicated speculation support to match the performance achieved by compile-time techniques (e.g., last_send).

Register communication scheduling, through code motion and loop restructuring, is successful in improving performance modestly for the integer benchmarks and significantly for the floating point benchmarks. Similar improvements are seen for both in-order and out-of-order PUs, indicating that reducing inter-task data dependence delays is important even for latency-tolerant PUs. Since the floating point benchmarks contain mostly loop-body tasks, they are benefited largely by loop restructuring, indicating that computation in loop bodies are dependent on loop induction variables. But the integer benchmarks contain mostly non-loop-body tasks, leading to less significant contribution from loop restructuring. While loop restructuring always moves loop induction variables to the top of their loop bodies, code motion is often constrained by intra-task dependences (control flow and data) when moving producers up and consumers down. Since the integer benchmarks use code motion more than loop restructuring, their improvements are modest. Register communication strategies and register communication scheduling grow in importance for larger number of PUs.

Lastly, dead register analysis significantly reduces register communication traffic, indicating that most register values are created and consumed within a few successor tasks, if not the same task. By including register dependences within a task (i.e., the register value involved in the dependence is dead beyond the task), the task selection heuristics create more opportunity for dead register analysis. Hence the combination of the heuristics and the analysis work together well. An implication of dead register optimization for the hardware is that a less aggressive hardware communication ring with modest bandwidth may suffice, if supplemented with dead register optimization.

6.3 Future work

Based on the experience gained from experimenting with compiler techniques, I suggest the following hardware and compiler optimizations to be tried in the future:

Larger tasks may be needed to increase the amount of parallelism exploited in the Multiscalar architecture. The main difficulty with large tasks is that the cost of squashes (both control flow and data dependence) increases with task size. Large tasks discard significant amounts of computation on squashes and lose performance due to both opportunity-costs and roll-back of useful computation. Hardware support for selectively squashing instructions dependent on misspeculated computation, instead of entire tasks may alleviate such misspeculation penalty. If selective squashing is supported in the hardware, the compiler may be able to employ more aggressive speculation to select larger tasks. It is important to note that if selective squashing aggravates PU critical paths then the gains obtained by improved squashes may be offset by clock speed considerations.

If task selection heuristics select tasks by including data dependence chains within tasks then load imbalance may occur due to inherent variations in the amount of computation done by different dependence chains. Hardware support for load balancing by either multiplexing multiple tasks on one PU simultaneously or starting the next task before the previous task commits by buffering state separately, may be important. Improvements gained by hardware load balancing may be offset by the increase in circuit complexity of PUs, which may have to maintain state of multiple tasks simultaneously. Thus, load balancing may result in a trade-off of parallelism for clock speed.

Static memory disambiguation combined with interprocedural analysis may improve task selection and inter-task data communication. Similar to register dependences, memory dependences may be included within tasks to avoid inter-task data communication or ARB misspeculations. Although memory dependences are dynamically synchronized, it may be beneficial to schedule any compile-time known memory dependences, similar to register dependences. Dynamic synchronization avoids only misspeculation squashes; any delay from producer of memory value to consumer of memory value may result in performance loss. Scheduling the dependences may help reduce such delays. Apart from scheduling, known memory dependences may be synchronized via compiler allocated synchronization resources, instead of dynamic synchronization resources (e.g., memory dependence prediction table entries), which may be more scarce.

This thesis has laid down the foundational infrastructure for studying the problem of compiling for the Multiscalar architecture. Many techniques employing varying degrees of cooperation between the hardware and the compiler to unleash the full potential of the Multiscalar architecture are emerging. Several other projects including the MIT RAW [107], the Stanford Hydra [79], and the CMU STAMPede [99] have adopted the approach of the Multiscalar architecture and have set to explore the numerous opportunities presented by the Multiscalar architecture through hardware and compiler techniques. In the future, I envision commercial Multiscalar processors overlapping execution of thousands of instructions to achieve orders of magnitude more performance than current microprocessors.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] V. H. Allan, J. Janardhan, R. M. Lee, and M. Srinivas. Enhanced region scheduling on a program dependence graph. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 72–80, Portland, OR, Dec. 1992. Association for Computing Machinery.

[3] F. Allen and J. Cocke. A program data flow analysis procedure. *Journal of the ACM*, 19(3):137–147, Mar. 1976.

[4] J. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[5] R. Allen and S. Johnson. Compiling c for vectorization, parallelization, and inline expansion. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 23, pages 241–249, Atlanta, GA, June 1988.

[6] S. P. Amarasinghe, J. M. Anderson, C. S. Wilson, S.-W. Liao, B. R. Murphy, R. S. French, M. S. Lam, and M. W. Hall. Multiprocessors from a software perspective. *IEEE Micro*, pages 52–61, June 1996.

[7] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computer Conference*, pages 483–485, 1967.

[8] G. Amdahl et al. Architecture of the IBM system/360. *IBM Journal of Research and Development*, pages 87–101, Apr. 1964.

[9] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the 5th ACM Symposium on Symposium on Principles and Practice of Parallel Programming*, pages 166–178, 1995.

[10] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–125, 1993.

[11] T. Asprey et al. Performance features of the pa7100 microprocessor. *IEEE Micro*, pages 22–35, June 1993.

[12] J. Barth. An interprocedural data flow analysis algorithm. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 119–131, Jan. 1977.

[13] J. Barth. A practical interprocedural data flow analysis algorithm. *Journal of the ACM*, 21(9):724–736, Sept. 1978.

[14] W. Baxter and H. B. III. The program dependence graph and vectorization. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, Austin, TX, Jan. 1989.

[15] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Conference Record of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 26, pages 241–255. Association for Computing Machinery, June 1991.

[16] M. T. Bohr. Interconnect scaling - the real limiter to high performance ULSI. *Solid State Technology*, pages 105–111, Sept. 1996.

[17] D. Bradlee, S. Eggers, and R. Henry. Integrating register allocation and instruction scheduling for riscs. In *Conference Proceedings of the Fourth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, Santa Clara, CA, Apr. 1991.

[18] S. Breach, T. Vijaykumar, and G. Sohi. The anatomy of the register file in a multiscalar processor. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 181–190, San Jose, CA, Nov. 1994. Association for Computing Machinery.

[19] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[20] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Conference Record of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 23, pages 47–56, Atlanta, GA, June 1988.

[21] B. Case. *Intel reveals Pentium Implementation Details*. Microprocessor Report, Mar. 1993.

[22] P. Chang, S. Mahlke, W. Chen, N. Warter, and W. Hwu. Impact: An architectural framework for multiple-instruction-issue processors. In *Conference Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275. Association for Computing Machinery, May 1991.

[23] D. Chen and P. Yew. Statement re-ordering for doacross loops. *Preceedings of the 1994 Annual International Conference on Parallel Processing*, 2:24–28, 1994.

[24] A. Chow and A. Rudmik. The design of a data flow analyzer. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, volume 17, pages 106–113, June 1982.

[25] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, volume 19, pages 247–258, June 1984.

[26] R. Cytron. Doacross: Beyond vectorization for multiprocessors. *Preceedings of the 1986 Annual International Conference on Parallel Processing*, pages 836–845, 1986.

[27] R. Cytron. Limited processor scheduling for doacross loops. *Preceedings of the 1986 Annual International Conference on Parallel Processing*, pages 226–234, 1987.

[28] D. Dhamdhere. Practical adaptation of the global optimization algorithm of morel and renvoise. *ACM Trans. Prog. Lang. Syst.*, 13(2):291–294, Apr. 1991.

[29] K. Diefendorff and M. Allen. Organization of the motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12, Apr. 1992.

[30] K.-H. Drechsler and M. Stadel. A solution to a problem with morel and renvoise's 'global optimization by suppression of partial redundancies'. *ACM Trans. Prog. Lang. Syst.*, 10(4):635–640, Oct. 1988.

[31] D.W.Anderson, F. Sparacio, and R. Tomasulo. The IBM system/360 model 91: Machine philosphy and instruction-handling. *IBM Journal of Research and Development*, pages 8–24, Jan. 1967.

[32] K. Ebcioglu. A compilation technique for software pipelining of loops with conditional jumps. In *Conference Record of the 20th Annual International Symposium on Microarchitecture*, pages 69–79, Colorado Springs, CO, Dec. 1987. Association for Computing Machinery.

[33] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Yale University, Feb. 1985.

[34] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30:478–490, July 1981.

[35] J. Fisher and B. Rau. Instruction-level parallel processing. *Science*, pages 1233–1241, Sept. 1991.

[36] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, Nov. 1993.

[37] M. Franklin. *The Multiscalar Architecture*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, Nov. 1993.

[38] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Conference Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67. Association for Computing Machinery, May 1992.

[39] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Conference Record of the*

25th Annual International Symposium on Microarchitecture, pages 236–245, Portland, OR, Dec. 1992. Association for Computing Machinery.

[40] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[41] P. Gibbons and S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Conference Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, volume 21, July 1986.

[42] B. A. Gieseke et al. A 600 mhz superscalar risc microprocessor with out-of-order execution. In *Preceedings of 1997 IEEE International Solid-State Circuits Conference*, pages 176–177, 1997.

[43] J. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Conference Proceedings of the 1988 International Conference on Supercomputing*, pages 442–452, St. Malo, France, July 1988.

[44] G. Grohoski. Machine organization of the ibm risc system/6000 processor. *IBM Journal of Research and Development*, 34(1):37–58, Jan. 1990.

[45] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, Apr. 1990.

[46] L. Gwennap. *Digital leads the pack with the 21164*. Microprocessor Report, Sept. 1994.

[47] L. Gwennap. *PowerPC 604 powers past Pentium*. Microprocessor Report, Apr. 1994.

[48] R. Hank, S. Mahlke, R. Bringmann, J. Gyllenhaal, and W.-M. Hwu. Superblock formation using static program analysis. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 247–255, Austin, TX, Dec. 1993. Association for Computing Machinery.

[49] R. Hanxleden and K. Kennedy. Give-n-take - a balanced code placement framework. In *Conference Record of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 107–120. Association for Computing Machinery, June 1994.

[50] M. Hecht and J. Ullman. A simple algorithm for global data flow analysis problems. *SIAM Journal of Computing*, 4(4):519–532, Dec. 1975.

[51] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for fortran d on mimd distributed-memory machines. In *Conference Proceedings of the International Conference on Supercomputing*, July 1992.

[52] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[53] P.-T. Hsu. Design of the R8000 microprocessor. *IEEE Micro*, pages 23–33, Apr. 1994.

[54] P.-T. Hsu and E. Davidson. Highly concurrent scalar processing. In *Conference Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 386–395. Association for Computing Machinery, June 1986.

[55] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control flow speculation in multiscalar processors. In *Conference Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 218–229, Feb. 1997.

[56] Q. Jacobson, E. Rotenberg, and J. Smith. Path-based next trace prediction. In *Conference Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 14–23. Association for Computing Machinery, Dec. 1997.

[57] S. Jain and C. Thompson. An efficient approach to data flow analysis in a multiple pass global optimizer. In *Conference Record of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 23, pages 154–163, Atlanta, GA, June 1988.

[58] M. Johnson. *Superscalar Microprocessor Design*. P T R Prentice-Hall, Inc., 1991.

[59] N. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, Albuquerque, NM, Jan. 1982.

[60] J. Kam and J. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[61] D. Kerns and S. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Conference Record of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–289. Association for Computing Machinery, June 1993.

[62] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, 1981.

[63] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218. Association for Computing Machinery, 1981.

[64] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Conference Record of the 1990 ACM SIGPLAN Conference on Pro-*

gramming Language Design and Implementation, pages 318–328. Association for Computing Machinery, June 1988.

[65] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of block algorithms. In *Conference Proceedings of the Fourth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, Apr. 1991.

[66] S. Mahlke, W. Chen, W. Hwu, B. Rau, and M. Sclansker. Sentinel scheduling for VLIW and superscalar processors. In *Conference Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 238–247. Association for Computing Machinery, Oct. 1992.

[67] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, Portland, OR, Dec. 1992. Association for Computing Machinery.

[68] E. P. Markatos and T. J. LeBlanc. Load balancing vs. locality management in shared-memory multiprocessors. Technical Report TR 399, Oct. 1991.

[69] T. Marlowe and B. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 184–196, San Francisco, CA, Jan. 1990.

[70] S. McFarling. Combining branch predictors. Technical Report TR-36, DEC-WRL, June 1993.

[71] S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 55–71, Portland, OR, Dec. 1992. Association for Computing Machinery.

[72] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, Feb. 1979.

[73] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[74] S. Muchnick and N. Jones. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[75] E. Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, Jan. 1981.

[76] A. Nicolau. Percolation scheduling: A parallel compilation technique. Technical Report TR-85-678, Cornell University, 1985.

[77] R. Oehler and R. D. Groves. IBM RISC system/6000 processor architecture. *IBM Journal of Research and Development*, 34(1):23–36, Jan. 1990.

[78] K. Olukotun, B. A. Nayfeh, L. Hammond, K. W. n, and K.-Y. Chang. The case for a single-chip multiprocessor. In *Conference Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Oct. 1996.

[79] J. Oplinger et al. Software and hardware for exploiting speculative parallelism in multiprocessors. Technical Report CSL-TR-97-715, Stanford University, 1997.

[80] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[81] D. Pnevmatikatos, M. Franklin, and G. S. Sohi. Control flow prediction for dynamic ILP processors. In *Conference Record of the 26th Annual International Symposium on Microarchitecture*, pages 153–163, Austin, TX, Dec. 1993. Association for Computing Machinery.

[82] D. Pnevmatikatos and G. Sohi. Guarded execution and branch prediction in dynamic ilp processors. In *Conference Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 120–129. Association for Computing Machinery, Apr. 1994.

[83] B. Rau, M. Schlansker, and P. Tirumalai. Code generation schema for modulo scheduled loops. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 158–169, Portland, OR, Dec. 1992. Association for Computing Machinery.

[84] B. Rosen. High-level data flow analysis. *Communications of the ACM*, 20:712–724, 1977.

[85] B. Rosen. Data flow analysis for procedural languages. *Journal of the ACM*, 26(2):322–344, Apr. 1979.

[86] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Conference Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–147. Association for Computing Machinery, Dec. 1997.

[87] R. M. Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978.

[88] B. Ryder. Incremental data flow analysis. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 167–176, Austin, TX, Jan. 1983.

[89] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Conference Proceeedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 192–201. Association for Computing Machinery, 1986.

[90] J. E. Smith. A study of branch prediction strategies. In *Proc. 8 Annual Symposium on Computer Architecture*, pages 135–148, May 1981.

[91] J. E. Smith. Dynamic instruction scheduling and the astronautics zs-1. *IEEE Computer*, pages 21–35, July 1989.

[92] M. Smith, M. Horowitz, and M. Lam. Efficient superscalar performance through boosting. In *Conference Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 248–259. Association for Computing Machinery, Oct. 1992.

[93] M. Smith, M. Lam, and M. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Conference Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354. Association for Computing Machinery, May 1990.

[94] J. Smith et al. The zs-1 central processor. In *Conference Proceedings of the Second International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 199–204, Oct. 1987.

[95] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Conference Proceedings of the 22nd Annual International Symposium on Computer Architecture*. Association for Computing Machinery, June 1995.

[96] G. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Conference Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, Santa Clara, CA, Apr. 1991.

[97] G. S. Sohi. Instruction issue logic for high performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, Mar. 1990.

[98] SPEC newsletter, Aug. 1995.

[99] J. G. Steffan and T. C. Mowry. The potential for thread-level data speculation in tightly-coupled multiprocessors. In *To appear in the Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Feb. 1998.

[100] J. Thornton. *Design of a Computer—The Control Data 6600*. Scotts, Foresman and Co., 1970.

[101] J. E. Thornton. Parallel operation in the control data 6600. *Fall Joint Computers Conference*, 26:33–40, 1961.

[102] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, Jan. 1967.

[103] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous

multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[104] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dy namic code sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 1–12, June 1997.

[105] T. Vijaykumar and G. S. Sohi. Compiling register communication in the multiscalar architecture. Technical Report CS TR-1333, University of Wisconsin, Madison, Nov. 1996.

[106] T. Vijaykumar and G. S. Sohi. Register communication strategies for the multiscalar architecture. Technical Report CS TR-1334, University of Wisconsin, Madison, Dec. 1996.

[107] E. Waingold et al. Baring it all to software: Raw machines. *Computer,* 30(9):86–93, Sept. 1997.

[108] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering,* 10(4):352–357, July 1984.

[109] S. Weiss and J. Smith. *Power and PowerPC: Principles, Architecture, Implementation*. Morgan Kaufmann Publishers, 1994.

[110] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. In *Conference Record of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 29, pages 31–37. Association for Computing Machinery, Dec. 1994.

[111] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Conference Record of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 26, pages 30–44, Toronto, Ontario, Canada, June 1991.

[112] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1990.

[113] K. Yeager. MIPS R10000 superscalar microprocessor. *IEEE Micro,* Apr. 1996.

[114] T.-Y. Yeh and Y. Patt. Two-level adaptive branch prediction. In *Conference Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61. Association for Computing Machinery, Nov. 1991.

[115] T.-Y. Yeh and Y. Patt. Alternative implementations of two-level adaptive training branch prediction. In *Conference Proceedings of the 19 Annual International Symposium on Computer Architecture*. Association for Computing Machinery, May 1992.

[116] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, NY, 1991.