The Dissertation Committee for Ramadass Nagarajan
certifies that this is the approved version of the following dissertation:

Design and Evaluation of a Technology-Scalable Architecture for Instruction-Level Parallelism

Committee:

Douglas C. Burger, Supervisor

Michael D. Dahlin

Stephen W. Keckler

Kathryn S. McKinley

Gurindar S. Sohi

Design and Evaluation of a Technology-Scalable Architecture for Instruction-Level Parallelism

by

Ramadass Nagarajan, B.Tech., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2007

To my parents and sisters.

Acknowledgments

I thank the members of my committee, Doug Burger, Mike Dahlin, Steve Keckler, Kathryn McKinley, and Guri Sohi for supervising this dissertation. I value their comments and suggestions that have improved its quality. I especially thank Guri Sohi for taking the time off his busy schedule to come down to Austin and serve on my committee.

I am eternally grateful to Doug Burger who has been my principal advisor and mentor through all the years in graduate school. Doug has been as perfect an advisor as one could ever hope for. I thank him for all the advice—technical, professional, and otherwise, hand-holding when necessary and cutting me loose at other times, being patient when I was tardy or sloppy, propping me up on opportune occasions, and of course, supporting me financially, even if it meant going to loggerheads with the establishment. His infectious enthusiasm and sharp wit only made the experience all the more enjoyable.

I thank Steve Keckler for serving as my de-facto co-advisor throughout my stay in school. Steve has complemented Doug in an advisory capacity in many ways, and I have benefited immeasurably from it. His insightful opinion, wise counsel, and deep intuition have been a guiding influence on my development as a researcher. Steve was the first faculty member that I met after I was admitted to the doctoral program at UT and he offered me the opportunity to explore architecture research during my first year. It was that experience and his advice that drew me to the

CART group, which he co-leads with Doug, and for that I will be forever grateful.

I thank Kathryn McKinley for several years of research collaboration on the TRIPS project. I thank her for the numerous interactions and thoughtful advice on technical writing and presentation. I cannot thank Doug, Steve, and Kathryn enough for their vision, leadership, and drive that set the direction of the TRIPS project in general, and this dissertation in particular.

I also thank Daniel Jimenez, Calvin Lin, and Chuck Moore for the research collaborations on the TRIPS project.

I thank the entire TRIPS team that helped transform an abstract research idea into working silicon. I would like to thank Robert McDonald for instilling rigor and attention to detail on every aspect of the TRIPS prototype implementation. I thank the entire hardware team—Raj Desikan, Saurabh Drolia, Madhu S. Sibi Govindan, Divya Gulati, Paul Gratz, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Karu Sankaralingam, Simha Sethumadhavan, and Premkishore Shivakumar—and the entire software team—Jim Burrill, Katie Coons, Mark Gebhart, Madhavi Krishnan, Sundeep Kushwaha, Bert Maher, Nick Nethercote, Behnam Robatmili, Sadia Sharif, Aaron Smith, and Bill Yoder—for their contributions. Without their collective contributions, the infrastructure necessary for this dissertation would not have been possible.

Karu Sankaralingam merits my special gratitude for his close collaboration. I started working with him on the PowerPC port of the SimpleScalar simulator in my first year, and the collaboration continued through the implementation of the TRIPS prototype processor. I have always admired his expansive scholarship, deep insight, and unique perspective on various issues, research and otherwise. I will forever cherish our initial joint work on the TRIPS architecture, which included non-stop brainstorming sessions, simulator hacking, and co-authorship of multiple TRIPS-related publications. Karu was also a constant source of entertainment around the

cubicle area that brought comic relief to the occasional dreary moment.

I thank Changkyu Kim for adding the TRIPS port to the *tsim-flex* simulator, which is one of the simulators that I used to generate the results presented in this dissertation. I thank Nitya Ranganathan for help with the SimPoint simulations.

I thank Heather Hanson, Divya Gulati, Madhu S. Sibi Govindan, and Aaron Smith for proof-reading portions of this dissertation.

I thank all the past and current members of the CART group for numerous technical discussions and feedback on papers and practice talks. I have always appreciated, and often dreaded, the comments and the questions that they threw my way, and the experience has made me a better researcher. The members of the CART group and the Speedway group have also provided a welcome distraction from the rigors of academic life through numerous social occasions.

I thank M.S. Hrishikesh for his patience and counsel during the numerous times I have turned to him for advice. He hosted me during my visit to Intel, Bangalore and has offered valuable feedback on my post-school career choices.

I also thank Kartik Agaram, for tolerating me as his roommate during the first few months, Simha Sethumadhavan, for enduring me as his cubicle mate for several years, Heather Hanson for her lessons on power and general advice, and Madhu S. Sibi Govindan and Suriya Subramanian for their valuable help at critical junctures.

I thank the staff of the Computer Sciences department, and in particular, Gloria Ramirez for her help as the graduate coordinator and the entire gripe staff for the timely resolution of my complaints with the facilities. I also thank Gem Naivar for her infinite patience, getting me paid on time, and help with travel arrangements and other miscellaneous paperwork.

I thank all friends outside the immediate research group for their merry company, numerous hiking, camping, and ski trips, and cooking me several delicious

meals. In particular, I would like to thank Murali Narasimhan, Karthik Subramanian, Siddarth Krishnan, Shobha Vasudevan, Ravi Karthik, Kunal Punera, Vinod Viswanath, Silpa Sigireddy, Bharat Chandra, Ramsundar Ganapathy, Subbu Iyer, Vishv Jeet, Prem Melville, Joseph Modayil, Aniket Murarka, Amol Nayate, Arun Venkatramani, Vibha volunteers, and the entire "Far West" gang.

Last, but not the least, I remain forever indebted to my parents and sisters. I cannot thank them enough for supporting me unflinchingly, continuing to worry for my well-being, and tolerating my extended absence from home for several weeks, months, and years at a stretch ever since high school. I especially thank my mom for asking me when I was going to graduate every time I spoke with her, and my dad and sisters for never asking me the same question. Without their constant encouragement and understanding I could not have come this far. To you, Mom, Dad, Hema, and Lalitha, I owe everything.

Ramadass Nagarajan

The University of Texas at Austin
May 2007

Design and Evaluation of a Technology-Scalable Architecture for Instruction-Level Parallelism

Publication No. _____

Ramadass Nagarajan, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Douglas C. Burger

Future performance improvements must come from the exploitation of concurrency at all levels. Recent approaches that focus on thread-level and data-level concurrency are a natural fit for certain application domains, but it is unclear whether they can be adapted efficiently to eliminate serial bottlenecks. Conventional superscalar hardware that instead focuses on instruction-level parallelism (ILP) is limited by power inefficiency, on-chip wire latency, and design complexity. Ultimately, poor single-thread performance and Amdahl's law will inhibit the overall performance growth even on parallel workloads. To address this problem, we undertook the challenge of designing a scalable, wide-issue, large-window processor

that mitigates complexity, reduces power overheads, and exploits ILP to improve single-thread performance at future wire-delay dominated technologies.

This dissertation describes the design and evaluation of the TRIPS architecture for exploiting ILP. The TRIPS architecture belongs to a new class of instruction set architectures called Explicit Data Graph Execution (EDGE) architectures that use large dataflow graphs of computation and explicit producer-consumer communication to express concurrency to the hardware. We describe how these architectures match the characteristics of future sub-45 nm CMOS technologies to mitigate complexity and improve concurrency at reduced overheads. We describe the architectural and microarchitectural principles of the TRIPS architecture, which exploits ILP by issuing instructions widely, in dynamic dataflow fashion, from a large distributed window of instructions.

We then describe our specific contributions to the development of the TRIPS prototype chip, which was implemented in a 130 nm ASIC technology and consists of more than 170 million transistors. In particular, we describe the implementation of the distributed control protocols that offer various services for executing a single program in the hardware. Finally, we describe a detailed evaluation of the TRIPS architecture and identify the key determinants of its performance. In particular, we describe the development of the infrastructure required for a detailed analysis, including a validated performance model, a highly optimized suite of benchmarks, and critical path models that identify various architectural and microarchitectural bottlenecks at a fine level of granularity.

On a set of highly optimized benchmark kernels, the manufactured TRIPS parts out-perform a conventional superscalar processor by a factor of 3× on average. We find that the automatically compiled versions of the same kernels are yet to reap the benefits of the high-ILP TRIPS core, but exceed the performance of the superscalar processor in many cases. Our results indicate that the overhead of

various control protocols that manage the overall execution in the processor have only a modest effect on performance. However, operand communication between various components in the distributed microarchitecture contributes to nearly a third of the execution cycles. Fanout instructions, which are necessitated by limited, fixed-width encoding in the dataflow instruction set, also contribute to non-trivial performance overheads. Our results point to an exciting line of future research to overcome these limitations and achieve low-overhead distributed dataflow execution.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Advances in semiconductor technology have spurred a remarkable growth in the capability of microprocessors. The first general-purpose programmable microprocessor was the 4-bit Intel 4004 and was used in calculators. It debuted in 1971 and consisted of approximately 2,300 transistors. Three and a half decades later, the mainstream Intel microprocessors have grown to contain approximately 300 million transistors. They integrate multiple 64-bit processor cores and several megabytes of on-chip memory on the same die, and run sophisticated operating systems, process rich streaming media, and even execute high-performance applications that traditionally have been in the domain of mainframe supercomputing systems.

Device scaling, architectural innovations, and cost benefits of a volume process have all contributed to this remarkable growth. Feature sizes have shrunk from 10 microns in 1971 to 65 nm in 2007, more or less in accordance with the famed Moore's law [103]. Production facilities for 45 nm devices are already on the horizon, and forecasts project continued geometry shrinks to at least 22 nm within the next decade [2]. Designers have rode this massive device scaling to reap performance improvements at the rate of nearly 50% per year from the late 1970's until the early part of this decade. Of late, this improvement has dropped to a more modest an-

nual growth of 20% due to limitations such as stopping clock rate growths and power constraints [70].

Much of the performance improvements in the early years resulted from exploiting the increased device density for wider datapaths and hardware support for memory management. Later, in the 1980's, microprocessors benefited from on-chip integration of several components: diverse computation logic, memory hierarchies, and pipelining. Since then, however, the bulk of the performance growth has come from increases in clock rates. Clock rates have improved due to equal contributions from both technology scaling and deeper pipelines, resulting in a 40% annual increase from 33 MHz in 1990 to over 2 GHz in 2001. Simultaneously microarchitectural techniques such as out-of-order execution, speculation, and cache prefetching have improved instruction-level parallelism to sustain the performance growth from increasing clock rates. But as recent trends indicate, the bullish performance ride on clock rate has abated. Power constraints [69, 157] and diminishing performance returns of deep pipelines [74] have ended the reign of clock rates in high performance microprocessors. Future improvements in performance must therefore come primarily from the exploitation of concurrency.

To extract concurrency, the microprocessor industry has shifted to multi-core designs and throughput computing. The reasons for this shift are mainly design simplicity, increased power efficiency, poor scalability of uniprocessor concurrency, and the relative ease of exploiting concurrency from data-parallel and thread-parallel workloads. Commercial dual-core chips debuted in 2001 [45], and recent trends point to a massive on-die proliferation of processor cores [18]. These approaches target application domains with explicitly parallel workloads, but neglect serial, irregular applications. Eventually, poor single-thread performance and Amdahl's law will inhibit the overall performance growth even on parallel workloads. Scaling single-thread performance is therefore important for the foreseeable future.

This dissertation presents a design and evaluation of a scalable uniprocessor solution for improving single-threaded performance. In the rest of this chapter, we present a brief overview of exploiting concurrency and the scalability problems of current solutions. We then present a high-level overview of the TRIPS architecture, which exploits concurrency using ultra-wide issue from a large distributed window of instructions. We then present the specific contributions of this dissertation and its relation to related work.

1.1 Exploiting Concurrency

Concurrency is the vehicle through which future microprocessors must attain performance improvements. Concurrency or parallelism is a property that determines how many independent operations can be performed by the hardware simultaneously. It exists in several forms—instruction-level parallelism (ILP), data-level parallelism (DLP), or thread-level parallelism (TLP). ILP results from executing multiple instructions from the same workload concurrently. DLP results from executing a single operation on multiple data concurrently. TLP results from executing multiple independent threads—different workloads, different transactions of a workload, or different threads of a parallel application—concurrently.

Historically, mainstream general-purpose processors have focused on ILP and exploited other forms of concurrency using extensions to the ILP hardware. For example, simultaneous multi-threading (SMT) uses ILP hardware to execute independent instructions from multiple threads in the same cycle [169]. The hardware for exploiting ILP dynamically consists of physical register files, register alias tables, issue windows, bypass networks, branch predictors, cache hierarchies, and prefetch engines. Exploiting higher ILP typically requires capacity growth in many of these structures, which engenders increased design complexity, power inefficiency, and poor delay scalability [102, 115]. Current processors already devote more than 85%

of the non-cache, non-TLB die area[1] to these structures [102], and further increases in their area are undesirable due to the above problems. The industry-wide response to this problem is a shift towards TLP and DLP—at the expense of ILP—and improving system throughput using single chip-multiprocessors (CMPs) [18,84,85,121].

The modular features of CMPs are attractive for reducing complexity, improving designer productivity, and increasing performance for a given power budget. These processors exploit the multiple cores to improve overall throughput in applications that exhibit explicit concurrency—internet transactions or parallel workloads. But serial portions of these workloads and single-threaded workloads utilize only one of the cores and are starved for performance, as each core itself sustains only modest ILP. Ultimately, Amdahl's law and stagnant clock frequencies would force designers to exploit higher single-threaded concurrency from the multiple cores, either by enhancing the ILP capability of the individual cores, or utilizing several of them to execute a single thread. Utilizing multiple cores to execute a single thread has been the focus of recent architecture research [15, 63, 67, 86, 113, 114, 159, 177], but effective solutions are still elusive. In this dissertation, we examine a single, but powerful core that exploits ILP without incurring the overheads of conventional superscalar cores.

The fundamental ILP scalability of superscalar cores stem from the over-reliance on large, centralized structures such as bypass networks, register files, and issue windows. Migration to wider issue architectures typically introduces multi-cycle wire delays, which increase the latency of critical performance loops—wakeup-select of back-to-back dependent instructions, register file access, and branch prediction—and degrade overall performance. Wire delays, in fact, have long been a problem for superscalar processors. In the late 1990's, the Alpha 21264 microarchitecture clustered the functional units, replicated the register files, and incurred a 1-cycle

[1]The remainder of the area is devoted to functional units.

inter-cluster latency to cope with global wire delays [79]. The Pentium 4 microarchitecture devoted two pipeline stages solely for transmitting global signals around the pipeline [73]. Future architectures must therefore address wire delays explicitly to improve performance.

The power inefficiency of superscalar architectures stem from the dynamic reconstruction of the program dataflow graph and per-instruction overheads. The instruction set architecture (ISA) conveys dependences to the hardware indirectly using register names, and the hardware dynamically identifies independent instructions using complex associative match logic. Every dynamic instruction must also access multiple power-hungry structures such as branch predictors, register alias tables, physical register files, and operand bypass networks. These structures typically account for more than a third of the total power consumption in modern processors [21] and inhibit the exploitation of further concurrency. Furthermore, the monolithic nature of the design increases the design and verification complexity significantly. Product cycles of 4 to 5 years involving several hundred designers are not uncommon in the industry. Future implementations must partition the global structures and limit their access on a per-instruction basis not only to enable modular design principles, but also to reduce overhead.

In summary, future ILP architectures must exhibit the following characteristics: a) high concurrency, b) complete distribution of hardware resources, c) power-efficient execution, and d) explicit optimization for wire delays. The design and evaluation of an architecture to attain these goals is the subject of this dissertation.

1.2 TRIPS: An EDGE Architecture

This dissertation presents the TRIPS architecture, which is an EDGE architecture, for exploiting high ILP at future, wire-delay dominated, VLSI technologies. Explicit Data Graph Execution (EDGE) architectures are a new class of instruction-set ar-

chitectures, in which the ISA aggregates large groups of instructions into program entities known as *blocks*. They convey the data dependences among instructions of the same block explicitly in the ISA, instead of using indirect register specifiers like conventional architectures. An EDGE ISA thus expresses the static dataflow graph directly to the hardware, instead of requiring it to rediscover data dependences dynamically at runtime.

The TRIPS architecture is an instantiation of an EDGE architecture. It partitions most of the traditionally centralized hardware structures, and exploits concurrency by issuing from a large window of in-flight instructions. It exploits the features of an EDGE ISA and a partitioned microarchitecture to address specific concerns of future high ILP architectures as follows:

Concurrent execution: The microarchitecture uses an array of concurrently executing ALUs, each of which operate only on a local window of instructions. It attains out-of-order execution using dataflow firing rules, wherein an instruction executes as soon as all of its operands are available. Aggregated across the entire array, the hardware offers a highly concurrent, out-of-order execution engine and a large window of in-flight instructions to exploit parallelism. A prototype implementation, in fact, supports 16-wide issue from a window of 1024 instructions.

Power efficiency: The architecture reduces many per-instruction overheads such as branch prediction, register renaming, operand bypass, associative issue window search, and register file access, which eliminates the need for most of the power-hungry structures present in a conventional superscalar processor. The explicit dependence encoding in the ISA expresses the dataflow graph directly to the hardware eliminating the need for dynamic rediscovery of the data dependences. It allows the hardware to directly forward the result of a computation to the consumer instructions using point-to-point communication without the use of inefficient centralized

structures such as register files, associative instruction schedulers, and ALU bypass buses. Across blocks, the architecture still performs conventional operations such as control-flow prediction, dynamic dependence checking, and register renaming. However, it amortizes these operations using large blocks, reducing the majority of the per-instruction overheads.

Design complexity: The microarchitecture embraces modular principles for managing design complexity and enhancing design productivity. It partitions most of the centralized structures such as data caches, register files, and issue windows into simple, replicated structures known as *tiles*. It connects the tiles using point-to-point networks where the links connect only nearest neighbors. Scaling to large systems is trivial; it involves additional replication and network connections, both of which require only modest design changes to microarchitecture.

Tolerance to wire delays: The microarchitecture avoids the scaling problems of global wire delays by using a tiled organization. It addresses the challenge of low overhead inter-tile communication using support from the compiler. The compiler determines placement labels for every instruction in the program, and the microarchitecture maps instructions on to the different tiles at runtime accordingly. Given a tiled organization, the compiler attempts to place instructions on tiles such that the physical distance that operands must travel *en route* to dependent instructions is minimized. The architecture thus couples compiler-driven instruction placement with hardware-driven execution order to mitigate communication delays and increase performance.

1.3 Thesis Statement

This dissertation addresses the design and implementation of a uniprocessor architecture for exploiting fine-grained concurrency with hardware structures distributed across the chip. It presents the architectural innovations that expose concurrency and the microarchitectural protocols that exploit the concurrency in a scalable, distributed fashion. It presents a detailed performance evaluation that demonstrates the benefits, overheads, and bottlenecks of single-threaded execution in a distributed microarchitecture.

This dissertation makes the following specific contributions:

ILP architecture: We describe EDGE ISAs that express dependences explicitly to the hardware. The TRIPS architecture, which is an EDGE architecture, enables a fine division of labor between the hardware and software for exploiting high concurrency and allows the reduction of several overheads present in conventional superscalar processors. The TRIPS microarchitecture integrates multiple, simple, heterogeneous components to implement a powerful uniprocessor. We describe the tile-level partitioning and the mechanisms for achieving scalable, wide-issue, out-of-order execution of single-threaded applications.

Hardware prototype implementation: We describe the TRIPS prototype processor, which is the first hardware implementation of the TRIPS architecture. In particular, we describe the design and implementation of the distributed control protocols that offer various services for execution, including fetch, flush, and commit. The TRIPS prototype chip is implemented in a 130 nm ASIC technology and consists of more than 170 million transistors. The chip contains two processors, each of which implements the TRIPS architecture, and a distributed 1 MB NUCA L2 cache [80]. At the time of writing this dissertation, the manufactured TRIPS

chips are fully operational and run at a clock frequency of 366 MHz.

Performance evaluation: We describe a detailed performance evaluation of the TRIPS architecture. We describe the mechanisms and methodology for evaluating the prototype implementation, including the development of a highly-optimized benchmark suite and the correlation of a detailed performance model with the hardware implementation. Analyzing a highly concurrent processor requires the development of sophisticated tools that identify the fine-grained interactions and bottlenecks among numerous inter-dependent microarchitectural events. This dissertation develops an efficient analysis based on critical path models to identify the bottlenecks of distributed execution.

Our results demonstrate that the largest overhead of distributed execution is the operand routing among the participating tiles in the microarchitecture. We observed that this overhead accounts for nearly a third and as much as 50% of the execution cycles. Our results illustrate that further research into creating and exploiting locality of communication is necessary for reducing the overhead of distributed execution. Fanout of operands to multiple consumers also presents some overheads, amounting to as much as 16% of the execution cycles. This result suggests a need for better ISA and microarchitecture support for wide broadcast of some operands.

Despite the overheads of distributed execution, our results, directly obtained from the TRIPS hardware, demonstrate that for a hand-optimized suite of benchmarks, the TRIPS processor attains a speedup ranging from 0.9 to 4.9 when compared to an Alpha 21264 core. On the same set of benchmarks, the TRIPS processor sustains good ILP which range from 1.1 to 6.5 instructions per cycle. The current compiled TRIPS code does not yet reap the full benefits of the high ILP TRIPS core, but does exceed the performance of the Alpha core on about half of the benchmarks we examined.

1.4 Dissertation Contributions

The design and development of the TRIPS architecture and the TRIPS prototype chip have been a collaborative effort involving many students, staff and faculty members at the University of Texas at Austin. In this section, I highlight my specific contributions and place it in the context of related work.

Karthikeyan Sankaralingam and I jointly proposed the high-level architectural and microarchitectural ideas, in association with our advisors Doug Burger and Stephen W. Keckler [109]. I explored the microarchitectural design space for exploiting ILP, including the necessary support for control speculation and predication, and the initial compiler algorithms for performing static instruction scheduling [108, 135]. I also developed various tools for performance analysis and identified bottlenecks for performance in the TRIPS architecture [107]. I designed and implemented portions of the TRIPS prototype chip. I led the design and implementation of the protocols and the global control logic, which provide various services for execution [136]. I also led the performance verification efforts for the TRIPS prototype processor. Chapter 4 provides a description of the implementation effort for the prototype chip and describes my specific contributions in greater detail.

The techniques developed during the research and implementation of the TRIPS architecture is the subject of multiple dissertations. My dissertation covers the architecture and microarchitecture for exploiting instruction-level parallelism and presents a detailed evaluation of various performance bottlenecks. Sankaralingam explores the techniques for exploiting data-level parallelism and describes the polymorphous capabilities of the architecture [132]. Other work has focused on developing efficient techniques for supporting distributed execution in the TRIPS architecture—Ranganathan (next-block predictor [126]), Liu (distributed instruction caches), Sethumadhavan (memory disambiguation [140]), Kim (non-uniform cache architectures [80]), and Gratz (on-chip networks [65, 66]).

1.5 Dissertation Layout

The rest of this dissertation is organized as follows. Chapter 2 presents recent, related work on exploiting ILP and compares it with the TRIPS architecture. In particular, it describes various tiled architectures, their execution models for exploiting ILP, and how they differ from the TRIPS architecture.

Chapter 3 presents the class of EDGE architectures, the instruction set model that exposes concurrency and allows distributed architectures to exploit ILP. It presents an overview of the TRIPS architecture, which is an EDGE architecture, the details of a microarchitecture implementation, and the rationale behind its design. It also presents the role of the compiler in the TRIPS architecture and the different optimizations it must perform to exploit high ILP.

Chapter 4 presents the details of the TRIPS prototype chip implementation. It describes the TRIPS prototype ISA, the processor microarchitecture, and the implementation of the protocols that offer various services for execution. It also describes the performance validation of the prototype processor and sets the context for understanding its performance. Finally, it describes the overall prototype implementation effort and highlights my specific contributions.

Chapter 5 presents the methodology for evaluating the performance of the TRIPS architecture. It describes the evaluation suite of benchmarks, the compilation infrastructure, and where necessary, the set of hand optimizations that were applied to improve code quality. It also describes the details of the performance simulators used for evaluating the architecture and the critical path analysis infrastructure used for identifying various performance bottlenecks.

Chapter 6 presents the results of our evaluation. It presents the raw performance results measured on the hardware. It reports the potential for high instruction throughputs in the TRIPS architecture and various overheads that inhibit parallelism. It also presents the relative effect of various microarchitectural bottle-

necks and suggests suitable enhancements for improving performance.

Finally, Chapter 7 presents a summary of the overall contributions of this dissertation and specific recommendations for future generations of the TRIPS architecture, in particular, and EDGE architectures, in general.

Chapter 2

Related Work

Superscalar processors exploit Instruction-Level Parallelism (ILP) in order to execute more than one instruction in a single clock cycle. They must deal with three common problems: instruction supply, which must provide an uninterrupted supply of instructions to execute, data supply, which must provide data just-in-time for the execution of the instructions, and dynamic instruction scheduling, which must analyze data dependences among a window of instructions and initiate the parallel execution of several independent instructions. Originally, the CDC 6600 and IBM System 360/91 machines used dynamic instruction scheduling mechanisms similar to modern superscalar processors, but were capable of executing only one instruction per cycle [10, 165]. Later-era microprocessors enhanced the superscalar capability by supporting dual issue of integer and floating point instructions [12], or using multiple integer and/or memory units [9]. Modern superscalar processors expand this capability significantly using extensive out-of-order execution for all instructions, support for precise exceptions, and speculation [117].

However, in recent years, superscalar processors have reached the limits of exploiting ILP. The out-of-order issue rate rarely exceeds four in current proces-

sors[1]—the latest Intel core microarchitecture issues only four instructions per cycle, and uses techniques such as macro-op fusion to increase the issue rate beyond four occasionally [85]. Untenable power constraints, high on-chip communication latencies, and increased design and verification costs have limited their scalability to wider issue. Designing a superscalar fabric that mitigates these constraints has been the focus of much research in both industry and academia. The proposed designs span the gamut between simple evolutionary techniques that reduce the complexity of dynamic schedulers and novel architectures that employ different execution models to improve ILP.

This chapter describes relevant work and compares it with the TRIPS architecture. First, it describes techniques that seek to reduce the complexity of various microarchitectural structures in dynamic superscalar processors, thereby improving their ILP scalability. Next, it describes alternative approaches that use simpler hardware by statically scheduling the ILP in a program. As CPU vendors are shifting entire design flows over to chip multiprocessors (CMPs), several researchers have started proposing techniques to exploit the available cores for improving ILP in a single program. We describe these techniques in Section 2.3. Finally, this chapter concludes with a survey of a few newly proposed architectures that employ unconventional execution models and microarchitectural organizations to exploit ILP.

2.1 Extending Superscalar Scalability

A typical superscalar microarchitecture consists of an in-order front-end, an out-of-order execution engine, and an in-order back-end [150]. The front-end predicts the program control flow, renames register operands in every instruction to eliminate false dependence hazards, and dispatches instructions into one or more issue queues

[1]The Alpha 21464 was designed to be 8-wide core, but was canceled during an advanced stage of development [44].

where the instructions wait until they execute. The out-of-order execution engine consists of the issue logic, functional units, and operand bypass networks. The issue logic buffers waiting instructions in issue queues and selects them for execution as and when operands become available. The functional units perform the actual execution and bypass networks forward the result operands to dependent instructions. The back-end logic commits the execution results to persistent architecture state in the same order intended by the program. The overall operation, however, involves several centralized hardware structures—physical register files, wakeup-select issue logic, load-store queues, and bypass networks—which exhibit poor power, latency, and area scalability at wider issue and wire-delay dominated technologies [6, 115]. Improving the scalability of these structures has therefore been the subject of much research effort. We survey relevant work in this section.

2.1.1 Issue Logic

The issue logic typically involves a tag broadcast for every result operand and an associative lookup of all instructions in the issue queue, and as such, is the one of the most complex structures in a superscalar processor. Several solutions have been proposed to mitigate its delay complexity. Prescheduling techniques attempt to schedule an instruction in advance by anticipating the time when it will be ready for execution [29, 30, 49, 99, 123]. This approach enables the hardware to consider only a subset of instructions for wakeup, thereby reducing latency. The proposed solutions differ largely on the mechanisms by which they deal with uncertain latencies induced by cache misses. Dependence tracking techniques maintain the dependences within the issue queue explicitly instead of using register tags [29,30,75,112]. When a register is produced, they use explicit producer-consumer links to minimize the number of instructions that must be searched for wakeup. Other techniques seek to reduce the power consumption in the issue queue by adjusting its size dynami-

cally [14, 26, 59]. Researchers have also proposed hierarchical queues [20], pipelined queues [22, 158] and segmented issue queues to permit fast clock rates [74, 123]. Finally, researchers have explored alternative circuit designs to improve the scalability of the issue logic [64, 72, 87].

2.1.2 Register File

Large multi-ported physical register files are necessary to eliminate false register dependences, store results of speculative operations, and support wide issue. The size of the register file and the number of ports together determine its scalability. Proposed solutions for improving the scalability typically attempt to minimize one or both components. Caching accelerates the access to commonly used registers. This technique uses a smaller, faster register file in front of a larger, slower register file to store frequently used values [19,25,40,176]. Banking sub-divides a larger register file into different banks that together provide the same capacity but decrease the size and ports for each bank [17, 40, 118, 168, 174]. Other solutions reduce the size of the register file by improving its utilization—delaying physical register assignment until execution or even writeback [101, 174]. Finally, some solutions reduce the number of ports by using auxiliary structures [81, 82, 118] or partitioning the registers into distinct read and write sets [142].

2.1.3 Load-Store Queues

Load-Store Queues (LSQs) buffer in-flight memory operations to dynamically disambiguate memory references. They typically involve an associate lookup structure that must be accessed for every load and store instruction. Their poor delay, power, and area scalability is one of the biggest impediments to wide superscalar execution. Improving LSQ scalability therefore has been an important focus of research in recent years. The proposed solutions attempt one or more of the following: a) reducing

the number of searches [139], b) eliminating associative operations [27, 61, 143, 161], and c) reducing the capacity of the LSQ [141]. We refer the reader to prior work for a concise description of the differences between various approaches [141].

2.1.4 Bypass Networks

The operand bypass network forwards result values from producer instructions to consumer instructions to eliminate pipeline bubbles in the execution of dependent instructions. The broadcast nature of this network inhibits its scalability to wider issue and wire delay dominated technologies [116]. To mitigate this problem, architects have proposed clustering [51, 79, 116, 130]. Clustering couples the issue queues and register files with the execution units and organizes them into different partitions. Operand forwarding within a cluster uses a bypass network and incurs zero-cycle delays, but between clusters it must incur additional delays. A centralized instruction steering logic directs instructions to different clusters and must balance the opposing goals of maximizing parallelism and minimizing inter-cluster communication. Clustering improves delay scalability as each cluster supports only smaller structures, but poor instruction steering can degrade performance significantly [51]. Tiled architectures described in the next section address the latter problem by using complete partitioning and an architecturally exposed routed interconnection networks.

2.1.5 Other Scaling Techniques

In addition to various scalability solutions for specific microarchitecture components, researchers have sought to improve overall performance by utilizing the existing resources efficiently. Non-blocking schedulers attempt to mitigate issue queue stalls resulting from long latency instructions and their dependents [73, 88, 104]. They migrate the stalling instructions from the issue queue to an auxiliary structure and re-

insert them at the appropriate time. This approach expands the window of in-flight instructions and exposes more parallelism. But it incurs the implementation complexity of large physical register files and re-order buffers. Checkpointing attempts to overcome this limitation [7, 39, 94]. This approach takes periodic checkpoints of the processor state and releases resources utilized by independent instructions following a stalled instruction after they have completed execution. Continual Flow Pipelines improve this mechanism further by releasing the resources held even by the dependent chain of stalled instructions and reclaiming them only when required [156].

2.1.6 Discussion

The TRIPS architecture takes a different approach to scalability. It is closest in principle to a clustered architecture, but supports more partitions to improve concurrency. It partitions all microarchitectural structures, including the instruction fetch logic and data memory into different tiles. It minimizes inter-tile communication latency using compiler support. It uses ISA support to eliminate dynamic scheduling hardware and mitigate the reliance on large unscalable structures.

2.2 Static Scheduling of ILP

VLIW architectures and their EPIC counterparts exploit parallelism by using extensive compiler support. They were originally proposed to exploit more parallelism using simpler hardware than the scalar machines of the day [56]. Fisher and Rau characterize them as independence architectures in which the compiler specifies which operations are independent of one another and orchestrates the entire execution of a program [57]. The compiler groups all independent operations into one long instruction indicating to the hardware that they must issue simultaneously. The hardware issues all these operations at the same time without performing any dependence or structural hazard checks. This approach reduces hardware complex-

ity compared to superscalar architectures.

The principal benefit from the VLIW approach to parallelism is the fact that the compiler can examine a large program window to discover independent operations. Using trace scheduling [55], the compiler can speculate on branches and migrate independent operations from distant program regions upwards. This ability mitigates Flynn's bottleneck [166] and offers the potential to increase parallelism significantly compared to scalar machines. However, poor tolerance to uncertain runtime latencies inhibits available parallelism. For example, in early VLIW machines, functional unit stalls for any one operation—such as divide—would suspend forward progress until that operation completed. The problem is more pronounced in the event of long cache misses, which would stall execution for several cycles, despite the availability of independent operations in future instructions. The performance losses from such uncertain dynamic events, disadvantages of code expansion, backward object code incompatibility, the emergence of RISC techniques, and increasing on-chip superscalar ability led to the gradual demise of VLIW machines except in specialized domains.

Superscalar machines increase performance by using branch prediction, which enhances the scope for parallelism, and dynamic scheduling, which offers the ability to overlap load miss latencies. Later-era EPIC machines attempted to obtain these benefits at the cost of increased hardware complexity [76, 96]. For example, the Itanium 2 processor includes hardware scoreboard logic to stall on operand use rather than operand creation to tolerate dynamic latencies [96]. Despite the additional hardware support, the processor cannot tolerate dynamic latencies as effectively as a superscalar processor.

The TRIPS architecture mitigates the disadvantages and combines the benefits of both VLIW and superscalar architectures. It uses ISA support to eliminate complex dynamic scheduling hardware. It exploits the compiler to schedule in-

structions for optimized communication, yet it retains the runtime flexibility of the superscalar hardware.

2.3 ILP from Chip Multiprocessors

Since the mid-1990's a number of research efforts have explored the option of using chip multiprocessors (CMPs) to improve single-threaded performance. These proposals fall under two broad categories—*thread-level speculation* and *pre-computation*. Thread-level speculation (TLS) uses multiple cores in a CMP to execute different portions of the same program concurrently and speculatively. Pre-computation uses the multiple cores to execute helper threads that enhance the performance of the main program. This section presents a brief overview of both approaches. For a comprehensive treatment of this subject we refer the reader to previously published work [62, 153, 160].

2.3.1 Thread-Level Speculation

Thread-level speculation (TLS) involves decomposing a single program into several small threads—either in the hardware or software—and spawning them in the hardware speculatively. The hardware usually includes support for detecting and recovering from any violation of program dependences that are caused by speculative execution. There have been a number of TLS schemes for CMPs [15, 63, 67, 86, 113, 114, 159, 177], and they find their roots in the Multiscalar work [152], which first proposed the creation of speculative tasks from a single program and executing them on different processing elements of a larger processor.

TLS proposals differ predominantly in their mechanisms for thread creation and communication of dependences among the threads. Most proposals spawn threads when the program control flow reaches a particular instruction—subroutine invocation, loop, or a control-flow split. For example, a TLS scheme may spawn

speculative threads for additional loop iterations, but commit them in program order after all dependence violations have been resolved. The thread spawns may be triggered automatically by the hardware or controlled by the compiler. Balakrishnan et al. observe that such control-flow mechanisms typically inhibit the ability to exploit far-flung parallelism [15]. Therefore, they advocate early speculative execution of many subroutines—triggered by the compiler—in parallel with other subroutines and the rest of the program. Other researchers advocate the use of transactional hardware and program annotations [68]. A sequential program written in a transactional language [31] decomposes the execution into several transactions, and the hardware executes many transactions concurrently.

2.3.2 Pre-computation

Pre-computation relies on the execution of an auxiliary program to aid the execution of the main program. The auxiliary program aids the main program by performing timely cache prefetches and resolving branches. Many of the techniques were developed in the context of SMT processors, but can also be suitably adapted for CMPs. The auxiliary program may be one of the following: a) a special program crafted by software [90, 131, 179, 180] and triggered for execution by the hardware at the opportune time, b) auxiliary program crafted entirely by the hardware [35, 50], c) continuation of the main program past a stalling event such as a L2 cache miss [16, 32, 47, 106], and d) second copy of the main program, which may execute incorrectly, but faster [122, 178]. Researchers have studied several techniques to improve the efficacy of the auxiliary program not only to generate useful and timely prefetches, but also to communicate useful pre-executed values to the main program and accelerate its execution.

2.3.3 Discussion

The recently taped-out "Rock" SPARC processor is purported to implement a form of run-ahead execution called Scouting [32, 167]. As CMPs continue to dominate the mainstream processor landscape, similar techniques will likely be adopted in the near future. Even so, it is unclear whether CMPs can exploit more than two cores to improve single thread performance. To exploit concurrency from several tens of cores that are projected to be integrated on a single chip within the next decade, many researchers are advocating a shift to parallel programming methodologies [11]. The TRIPS architecture takes a fundamentally different approach. It maps large units of computation from sequential programs on to multiple distributed processing tiles and allows the compiler to optimize for communication latencies. It uses several protocols to orchestrate the execution of a single program across the distributed substrate. The distributed tiles provide a large instruction window and wide execution bandwidth, which are exploited for high concurrency.

2.4 Extracting Concurrency through Tiling

In recent years, *tiling* has emerged as a microarchitectural technique for mitigating complexity and enhancing concurrency. It is a technique in which the entire processor microarchitecture is physically organized as a collection of numerous smaller replicated structures called *tiles* that are connected together using one or more interconnection networks. Each tile performs a small microarchitectural task and exchanges control and data information with other tiles using the interconnection network. The hardware exploits parallelism by partitioning the execution of a single program across several tiles and allowing the tiles to operate concurrently.

In principle, tiling dates back to the Multiscalar processors proposed by Sohi et al. [152]. A Multiscalar processor sequences through the control-flow graph

of a program in large, compiler-constructed units of work known as *tasks*. The processor speculatively executes several tasks in parallel on a collection of processing units that communicate with each other using an interconnection network. The hardware provides the necessary support to ensure the correct resolution of control and data dependences among the speculatively executing tasks. Since each task, which may include entire loop bodies and functions, may dynamically translate to potentially hundreds and even thousands of instructions, a Multiscalar processor has the ability to examine far-flung regions in the program simultaneously to exploit ILP. While Multiscalar architectures demonstrated the potential for exploiting high ILP, recently proposed tiled architectures extend the concept to address the latency of communication on a distributed hardware substrate explicitly.

The TRIPS architecture is an example of a tiled architecture in which several heterogeneous tiles comprise the entire processor. A number of other recent architectural proposals also employ tiling. Prominent among those are RAW [172], Smart Memories [93], WaveScalar [162], and Tartan [100]. The RAW architecture investigated several advantages and issues with tiling. Specifically, it investigated the characteristics of scalar operand transport networks [164] necessary for tiled architectures and techniques for partitioning code and data across several tiles [89]. This section provides a brief overview of various tiled architectures and compares them with the TRIPS architecture.

Tiled architectures can be categorized along three defining attributes: a) tile composition, b) connectivity, and c) execution model. Table 2.1 provides a summary of these attributes for various tiled architectures. Tile composition refers to the number and type of the individual tiles that comprise the entire processor. Connectivity refers to the organization of the interconnection networks that connect various tiles. Finally, execution model refers to how each architecture exploits concurrency and the division of labor between the compiler and the hardware. For example, TRIPS

Architecture	Tile composition	Connectivity	Execution model
TRIPS	heterogeneous multiple replicated tiles each implementing a different functionality—execution, data caches, instruction caches, and register files	dynamically routed mesh network for operands, simple point-to-point links for control	block-atomic, hybrid dataflow
RAW	homogeneous multiple identical tiles each implementing a complete processor with caches and and register files	two statically routed and two dynamically routed mesh networks for control and data	statically orchestrated MIMD
WaveScalar	heterogeneous groups of identical PEs coupled with caches to form a cluster that is replicated	hierarchical network—pipelined buses connect PEs and switched network connects the clusters	tagged-token dataflow
SC	heterogeneous identical PEs organized into pages and replicated to form a cluster, which is replicated to form the entire processor.	hierarchical network—statically routed within one cluster, dynamically routed mesh network between clusters	static dataflow

Table 2.1: Attributes for various tiled architectures.

uses a collection of heterogeneous tiles, a flat operand interconnection network in which every tile is a node on the network, and an execution model in which the compiler assigns placement labels for instructions and the hardware executes them dynamically in a dataflow fashion. In the sections below, we describes the attributes of various tiled architectures and how they compare with the TRIPS architecture.

2.4.1 The RAW Architecture

The RAW architecture consists of a homogeneous array of replicated tiles connected together using programmable point-to-point interconnects. It fully exposes all the hardware resources—including the tile components and the interconnects—to the compiler and allows it extract concurrency by precisely orchestrating the computation within each tile and the communication between different tiles [172]. Taylor

et al. describe a hardware implementation that consists of 16 tiles communicating using four flat two-dimensional mesh networks [163]. All tiles are identical and each contains a MIPS-like, single issue, in-order execution pipeline, a set of routers implementing various communication protocols, and caches for instructions and data.

The RAW compiler spatially distributes the instructions and data needed by a program across different tiles. The partitioning is done such that it maximizes both locality, which reduces communication latencies between producer and consumer instructions, and parallelism, which distributes independent instructions across different tiles [89]. Similar to a VLIW architecture, the compiler specifies the temporal execution order of instructions within each tile. In addition, it specifies the temporal order of communication events in each tile and the precise communication routes between instructions in different tiles. The hardware adheres to the statically specified order faithfully.

The principal difference between RAW and the TRIPS architectures is the agent through which each exploits concurrency. The RAW architecture entirely relies on the compiler, whereas TRIPS employs both the compiler and runtime hardware for exploiting concurrency. In RAW, the compiler discovers parallelism, partitions instructions and data among the tiles, performs static branch prediction, disambiguates memory references, and schedules communication routes for all statically determinable dependence paths. In TRIPS, the compiler expresses the dependence graph of computation explicitly to the hardware and partitions the instructions among several tiles to maximize concurrency and minimize communication overheads. However, the hardware is free to select the temporal order for both execution and communication, resulting in wide out-of-order execution.

The static orchestration of concurrency and communication in RAW offers a few benefits: a) with complete knowledge of the hardware configuration, the compiler can optimize for both locality of communication and concurrency in execution,

b) should the communication paths traverse multiple tiles, the compiler can pick the best routes by avoiding congested network nodes and links, and c) the absence of any dynamic scheduling logic reduces hardware complexity. However, it also suffers from a few disadvantages. First, it limits the scope of scheduling to only statically determinable dependences. If dependence paths cannot be statically determined, communication has to revert to a slow, fallback dynamic routing network. If a program has several such paths, performance degrades considerably. Second, dynamic events such as cache misses limit concurrency by stalling the execution of independent instructions within a single tile and potentially other tiles. Finally, it limits the hardware's ability to further improve concurrency through advanced speculation techniques such as dynamic branch prediction and dynamic memory dependence prediction, both of which have proven to be effective for traditional architectures [105, 175].

The TRIPS architecture reaps the same benefit as RAW with respect to the partitioning of a program's instructions across several tiles. The compiler estimates the temporal execution order on the hardware to optimize for communication latencies and concurrency, but does not convey any temporal constraints to the hardware. This feature allows the hardware to execute instructions dynamically as soon as their operands are available, providing a better tolerance for long latency events such as cache misses. The explicit encoding of dependences in the ISA reduces the complexity of dynamic scheduling typically seen in conventional superscalar processors. Furthermore, the absence of static temporal constraints permits the hardware to employ control speculation and memory dependence speculation to improve concurrency further. The lack of static temporal constraints, however, occasionally causes contention in the network links and degrades performance.

2.4.2 WaveScalar

WaveScalar is a tagged-token dynamic dataflow architecture proposed recently at the University of Washington [162]. Similar to TRIPS, it uses a dataflow instruction set. The compiler partitions a program into sets of instructions called *waves*. Each wave is an acyclic region of the program's control flow graph and may include an arbitrary number of control flow forks and joins. The compiler converts all control dependences to data dependences, however, using dataflow operators such as conditional splits.

The microarchitecture of a WaveScalar processor consists of a number of homogeneous processing elements (PEs) and data cache banks communicating using a hierarchical interconnection network. Each PE contains an instruction store to hold the instructions mapped to the PE, a matching table to hold the data tokens for the instructions, and logic to control the dataflow execution and communication. A specialized memory interface called *wave-ordered memory* uses compiler-specified sequence numbers to enforce correct program ordering of memory operations.

WaveScalar shares some common characteristics with the TRIPS architecture. Both architectures use a dataflow instruction set, which expresses the dependences among instructions explicitly. Both architectures rely on the compiler for good instruction placement—assignment of instructions to PEs—to reduce dynamic operand communication latencies. Both architectures employ compiler-specified sequence numbers—but use different hardware mechanisms—to order memory operations correctly, which is necessary for supporting programs written in imperative languages such a C/C++ and Java. Finally, a WaveScalar wave is similar to a TRIPS block and is amenable to all the dataflow optimizations developed for TRIPS blocks [149]. However, WaveScalar differs from TRIPS in significant ways, specifically the execution model, control flow implementation, and speculative execution.

Execution model: WaveScalar uses dataflow execution for the entire program, whereas TRIPS employs dataflow execution only among a bounded set of instructions. The execution of an instruction in WaveScalar is purely determined by dataflow communication. Between different waves, WaveScalar must use special instructions to manage dataflow tokens and transfer data values. The TRIPS architecture, however, transfers data operands between blocks using a register file. The hardware employs register renaming and dynamic register forwarding similar to traditional superscalar architectures to support concurrent execution of multiple blocks.

Control flow overheads: Dataflow architectures incur overheads of managing control flow within the program. For example, the WaveScalar compiler must convert all control dependences to dataflow using appropriate data steering instructions. While some optimizations are possible [120], these instructions still present considerable execution overheads and reduce achievable performance. For example, executing one fork of a branch often must wait for the resolution of the condition that defines the fork, which elongates the critical path of execution. On the other hand, executing both forks of a branch and selecting the appropriate output at the end results in wasteful execution that increases contention for shared hardware resources. Extended to an entire program, the overheads of polypath execution and dependence height extensions inhibit the performance of dataflow architectures considerably [23].

The TRIPS architecture incurs similar, but fewer overheads. Within a single TRIPS block, the compiler uses predication to convert control dependences to data dependences and various predication optimizations attempt to reduce the associated overheads [149]. Across multiple blocks, however, the compiler retains control dependences instead of converting them to data dependences. The hardware dynamically detects data dependences across multiple blocks and enables dataflow

"stitching" of these blocks without incurring any control-flow instruction overheads.

Speculative execution: Pure dataflow execution is not amenable to traditional speculation techniques such as control speculation and memory dependence speculation. The challenges of detecting mis-speculation and effecting a rollback recovery protocol have typically prevented dataflow architectures, including WaveScalar, from employing any form of speculation [23]. On the other hand, the block-atomic execution model makes TRIPS amenable to many forms of speculation. The boundaries of a block provide an architectural point to detect mis-speculations and rollback the execution state to the beginning of an incorrectly executed block. The TRIPS prototype processor employs both control speculation and memory dependence speculation to improve performance.

2.4.3 Spatial Computation

Spatial Computation (SC) is a model of computation optimized for wire delays [24]. In this model, programs written in high-level languages such as C are directly compiled down to hardware structures that are completely distributed, use only simple local control protocols, and operate without any global signals. In prior work, researchers have proposed two incarnations of SC, namely Application Specific Hardware (ASH) [24] and its extension, Tartan [100]. An integral component of both these architectures is an asynchronous hardware fabric that contains a multitude of heterogeneous functional units, each of which is statically synthesized for a single program operation and not shared with any other operations.

SC adopts a pure static dataflow execution model originally proposed by Dennis et al. [42]. Therefore, it exhibits all of the disadvantages of pure dataflow execution such as control flow overheads and lack of speculative execution similar to WaveScalar. In addition, its static dataflow nature prohibits SC from executing an instruction unless all consumers have sourced the result produced by the previous

instance of the same instruction. This limitation affects loop-level parallelism, as effectively only one iteration of a loop can be in execution at any instant. TRIPS, however, uses control speculation to execute several iterations concurrently and exploit loop-level parallelism.

2.4.4 Other Tiled Architectures

There have also been a number of other proposals for tiled architectures—Smart Memories [93], Vector-Threaded Architectures [83], Synchroscalar [111], and CDE [5]. The Smart Memories architecture uses a reconfigurable fabric consisting of homogeneous tiles and a hierarchical network, and is suited for exploiting all types of parallelism [93]. Vector-Threaded Architectures use multiple lanes of homogeneous processors connected by a unidirectional ring network and are specifically designed to improve performance on data-parallel workloads [83]. The Synchroscalar architecture consists of homogeneous tiles and a reconfigurable interconnect fabric and is tailored towards statically-orchestrated SIMD-style computation [111]. Finally, the CDE architecture consists of a number of homogeneous processors connected by a dynamic mesh network, and exploits both compiler support and runtime speculation to exploit ILP [5]. Unlike these architectures, TRIPS combines several heterogeneous tiles on an interconnection network to form one large uniprocessor that can be configured to exploit all types parallelism.

2.4.5 Discussion

A tiled microarchitecture offers a number of benefits. First, it enhances design scalability by recognizing and tolerating wire delays. By keeping each tile small, typically to a few mm^2 of area, signals within a tile need to traverse only small distances. Global communication among different tiles, however, is exposed at the architectural level and accomplished using point-to-point interconnection networks

and communication protocols, without the use of any global wires. Second, tiling enhances design productivity. Design complexity is limited to the design of a single tile—one for each type—and the interconnection network. The entire processor can then be constructed by merely replicating a single tile and connecting the replicas together. This modular organization lends to hierarchical implementation and verification methodology, easing the overall design effort considerably. Finally, tiling offers an easy migration path to larger architectures. Future generations can stamp out larger processors without significant rework by merely replicating the tiles and expanding the interconnection network to include more nodes.

2.5 Summary

In this chapter, we presented an overview of different approaches to exploiting ILP and compared them with the TRIPS architecture. We described several proposals that examined techniques to scale different portions of a traditional superscalar processor. We described approaches that seek to exploit the multiple processors in a chip-multiprocessor to improve the performance of a single thread of execution. Finally, we described in depth various recent proposals that organize the processor microarchitecture as replicated tiles to address the complexity and delay constraints of future technologies. The subsequent chapters describe the TRIPS approach to concurrency using ISA support, the compiler, and a tiled microarchitecture.

Chapter 3

TRIPS: An EDGE Architecture

The instruction set architecture (ISA) for a machine delineates the responsibilities of the hardware and the software, and aids the independent development of each. The most popular ISAs have enormous installed software bases, and therefore, ISA re-designs are only glacial in nature. To exploit new hardware capabilities, ISAs typically add only extensions [46, 119], and avoid the introduction of any new features that would mandate the recompilation of existing software. However, as the underlying silicon technology changes, existing ISAs must undergo suitable changes. Otherwise, they will inhibit the benefits offered by new technologies.

New hardware technologies, in fact, have spurred radical changes to ISAs in the past. When memory was at a premium, CISC ISAs featured dense encodings, variable instruction lengths, and few architected registers to reduce the program footprint in memory. As VLSI technology evolved, memory became cheaper and the number of on-chip transistors became a limiting resource. Therefore, RISC ISAs and their CISC equivalents of μ-ops were introduced—with simpler encodings, fixed-length instructions, and few instruction modes—to simplify the control logic, implement aggressive pipelining, and fit an entire microprocessor on a single chip. However, as VLSI technologies yield power and wire latency-dominated substrates,

pipeline-centric RISC ISAs are no longer viable options for high performance [74]. Microprocessor designs must turn to new ISAs that are both conducive to the expression of concurrency and amenable to communication-dominated execution for attaining high performance.

EDGE architectures are a new class of architectures intended for extracting high concurrency at future sub-45 nm CMOS technologies. This chapter describes the salient features of EDGE architectures and their differences from conventional architectures. It then describes the TRIPS architecture, which is an example of an EDGE architecture, and the architectural and the microarchitectural features that expose and exploit high concurrency. It also describes the responsibilities of the compiler, and shows how the division of labor between the compiler and the microarchitecture is a good match for future technologies. Along the way, this chapter also discusses various design alternatives and contrasts them with the particular design choices made in the TRIPS architecture.

3.1 EDGE Architectures

EDGE stands for Explicit Data Graph Execution and has two defining characteristics: *block-atomic execution* and *direct instruction communication*.

Block-atomic execution: An EDGE architecture aggregates a group of instructions into a single entity called a *block* that is treated as an atomic unit of execution. Just as a conventional architecture sequences through instructions, an EDGE architecture sequences through blocks. Logically, the runtime hardware fetches the instructions belonging to a single block *en masse*, maps them to the execution resources, executes the instructions, and commits their results in an atomic fashion. The hardware either commits the results of all instructions in the block or nullifies the execution of the entire block.

Direct instruction communication: An EDGE architecture permits the hardware to communicate the result of a producer instruction directly to the consumer instruction within the same block, instead of writing to an architected namespace such as the register file. The ISA provides support for specifying the consumers for an instruction directly, instead of specifying them indirectly using source register names. For example, consider the following RISC instruction that adds the values stored in registers `R1` and `R2` and stores the result in `R3`:

```
add R3, R1, R2
```

An EDGE architecture may encode the instruction as follows:

```
add T1, T2
```

The above encoding specifies the consumer instructions `T1` and `T2` directly with the `add` instruction, instead of expressing that data dependence through the register name `R3`. The source operands are not specified with the instruction. Instead, they will be specified by the producer instructions that target the `add` instruction. In this model, register data dependences can be expressed explicitly using target encoding. Memory dependences, however, must still be expressed using a shared namespace.

3.1.1 Advantages of EDGE Architectures:

An EDGE architecture provides an alternative interface between the compiler and the hardware compared to conventional architectures. Its two attributes of block-atomic execution and direct-instruction communication offer the following benefits:

Explicit conveyance of dependences: In a conventional out-of-order RISC (or CISC) architecture, the dataflow graph of execution known to the compiler is lost amidst the indirect register encoding of dependences. The hardware dynamically reconstructs the dependence graph using register alias tables and issue queues, and picks independent instructions to execute from potentially hundreds of instructions

in flight. VLIW architectures instead pack multiple independent operations within a single instruction. However, the statically orchestrated issue has a poor tolerance to dynamic latencies such as cache misses. EDGE architectures use the direct instruction communication to make the dataflow graph explicit and allow the hardware to execute each instruction in a dataflow fashion as soon as all of its operands are available. This feature provides dynamic issue, which helps tolerate long cache misses, and extricates the hardware from unscalable dynamic dependence check logic.

Reduction in per-instruction overheads: In conventional out-of-order processors, every instruction must traverse through multiple large structures for highly concurrent execution—register alias tables, issue queues, ALU bypass buses, and multi-ported register files. The latency and power dissipation in these structures scale poorly to smaller device widths, and inhibit the use of wide issue and large instruction windows [6, 115]. An EDGE architecture mitigates these problems. It allows the hardware to forward the result of an instruction directly to its consumer, thus triggering its execution without any associate issue queue lookups. Furthermore, it allows the hardware to route the result to its consumers using point-to-point communication instead of broadcast on a bypass bus. The block-oriented execution allows the architecture to avoid saving temporary result values within a block to a register file. It thus minimizes accesses of the register alias table and the register file to only the register inputs and outputs of a block.

Match for distributed architectures: The latency of communication between different processing elements affects the performance of a distributed architecture. EDGE architectures provide the opportunity to minimize communication latency. For example, mapping dependent instruction paths along short physical routes in the hardware and mapping independent instructions to different processing elements optimizes both concurrency and latency. The explicit expression of dependences in

the ISA permits the hardware to realize direct producer-communication on short paths using guidance from the compiler.

Division of labor between the compiler and hardware: EDGE architectures exploit the capabilities of both the compiler and the hardware for higher concurrency without the attendant inefficiencies of large structures required for out-of-order superscalar execution. They permit the compiler to focus on exposing concurrency through large blocks of instructions. The ISA expresses the concurrency to the hardware using dependence arcs and the hardware exploits the concurrency using low-overhead dataflow execution.

3.1.2 Discussion

The concept of block-atomic execution is not new; it was originally proposed in the form of block-structured ISAs by Melvin et al. to improve superscalar execution [97, 98]. EDGE architectures augment a block-structured ISA with direct instruction communication to reduce the overhead of dynamic superscalar execution. Atomic execution of instruction sequences also shares some similarities with checkpoint-based execution [7, 39, 94] and transactional execution [68]. Both techniques roll back the execution state of a group of instructions to recover from incorrect speculation. However, when speculation is correct, checkpoint-based mechanisms expose architecture state modified by individual instructions of a group, whereas transactional execution and EDGE architectures provide the semantics of "indivisible" group execution.

3.1.3 Implementation Choices

Given the two defining attributes of an EDGE architecture, numerous implementation choices are possible. The particular design points may vary based on the

composition of a block and the expression of inter-instruction dependences within the block. In the following paragraphs, we explore these two design axes.

Block composition: A block in an EDGE architecture could be any sequence of instructions—a basic block, a superblock [77], a hyperblock [92], or even a runtime trace of instructions [130]. Blocks may be of a fixed size or variable sizes and may include instructions from only one control path or multiple control paths. An architecture may also impose other constraints on the block composition, such as the maximum number of instructions, maximum number of load or store instructions, and the maximum number of inputs and outputs from the block.

Dependence encoding: An EDGE architecture may choose to express inter-instruction dependences in different ways. Similar to the example described in the previous sections, an architecture may directly encode the consumers with each instruction. Alternately, it may also use a special set of instructions to communicate the result to one or more consumers, similar to the RAW architecture. If it encodes consumers directly with an instruction, the architecture may choose fixed length instructions and restrict the number of consumers, or choose variable length instructions and encode all consumers.

The TRIPS architecture described in the next section is an example of an EDGE architecture. Recently proposed tiled architectures such as RAW [172] and WaveScalar [162] may also considered as EDGE architectures. The TRIPS architecture composes a block from a linear control sequence of instructions and encodes intra-block consumers directly with an instruction. The RAW architecture supports direct producer-consumer communication through special statically-scheduled instructions that specify the communication routes. WaveScalar encapsulates all control paths of a program region into a single group called a wave, and supports direct producer-consumer communication using dataflow arcs similar to the TRIPS

ISA.

3.2 The TRIPS Architecture

This section describes the TRIPS architecture, which is one possible instantiation of an EDGE architecture. It couples dynamic, hardware-controlled out-of-order execution with a compiler-ordained mapping of instructions. The compiler specifies implicit placement labels for instructions, and the hardware maps instructions to physical hardware locations accordingly. The ISA encodes each program as a sequence of large blocks, and instructions in each block explicitly encode the labels of intra-block consumer instructions. The hardware uses this explicit encoding for point-to-point data communication between producers and consumers, and performs dataflow execution of intra-block instructions. This section describes the features of the TRIPS architecture and how it supports a distributed microarchitecture for exploiting high concurrency.

3.2.1 TRIPS ISA

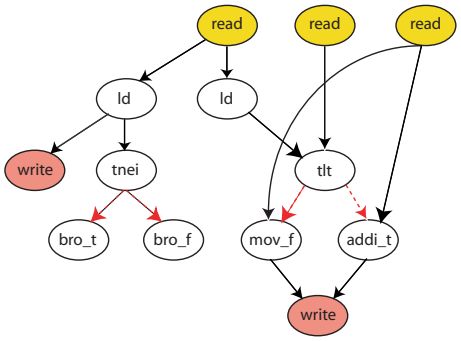
The key design decisions for an EDGE architecture are the block composition and the expression of inter-instruction dependences. As outlined in the previous section, numerous design choices exist for each. An ideal architecture is one that uses large blocks to reduce per-instruction overheads and uses few overhead instructions for managing control flow. The architecture must provide solutions for expressing dependences among the instructions of the same block and across multiple blocks with few overheads. We begin this section by illustrating the choices made in the TRIPS architecture. At the end of this section, we discuss the rationale behind its design features and possible alternatives.

Figure 3.1 shows a simple C program and its representation using the TRIPS ISA. The first portion shows the program, which traverses a linked list and counts

a) C code fragment

```
while (ptr != NULL) {
        if (x < ptr->data) {
                num++;
        }
        ptr = ptr->next;
}
```

c) Loop body DFG



b) TRIPS RISC-like instructions

```
begin blk0

read   $t0, $g0              ; ptr
read   $t1, $g1              ; x
read   $t2, $g2              ; num

ld     $t3, ($t0)            ; ptr->data
tlt    $t4, $t1, $t3         ; x < ptr->data
addi_t<$t4>    $t5, $t2, 1   ; num++
mov_f<$t4>     $t5, $t2
ld     $t6, 8($t0)           ; ptr->next
tnei   $t7, $t6, 0           ; ptr != 0
bro_t<$t7>   blk0            ; branch to block blk1
bro_f<$t7>   blk1            ; branch to block blk2

write  $g2, $t5               ; num
write  $g0, $t6               ; ptr

end blk0
```

d) TRIPS instructions

```
begin blk0

R[0] read G[0] N[1,L] N[0,L]
R[1] read G[1] N[5,L]
R[2] read G[2] N[2,L] N[9,L]

N[1] ld N[5,R]
N[5] tlt N[2,p] N[9,p]
N[2] addi_t 1  W[2]
N[9] mov_f  W[2]
N[0] ld 8 N[4,L]
N[4] mov W[0] N[8,L]          ; fanout
N[8] tnei 0 N[12,L]
N[12] mov N[16,p] N[6,p]      ; fanout
N[16] bro_t  blk0
N[6] bro_f  blk1

W[2] write G[2]
W[0] write G[0]

end blk0
```

Figure 3.1: TRIPS code example.

the number of elements greater than a given value `x`. The second portion shows the entire body of the while-loop accommodated in a single TRIPS block and represented using a RISC assembly language similar to MIPS. The annotations next to each instruction describe the program statement corresponding to that instruction. The block carries the label `blk0`, and the branches transfer control flow to the beginning of the same block or the beginning of another block labeled `blk1`. The third portion shows the dataflow graph of the block, and the last portion depicts the instructions encoded using the TRIPS ISA. We observe that the TRIPS ISA exhibits several key differences from its RISC counterpart: a) read instructions, b) write instructions, c) use of predication, d) fanout instructions, e) target encoding, and f) block format. The following paragraphs describes each of these differences in detail.

Read/Write instructions: Read instructions and write instructions specify the live register inputs and outputs of a block explicitly, instead of specifying them with other instructions. Read instructions specify the input registers for the block and the instructions that consume those values. For example, the instruction `read $t1, $g1` forwards the value of `x` saved in register `$g1` to the test instruction `tlt`. Similarly, the write instructions specify live registers written by the block. For example, the instruction `write $g0, $t6` consumes the result of the second `ld` instruction and saves it in the general register `$g0`, which corresponds to the live program variable `ptr`. By isolating register file accesses using read and write instructions, all other instructions strictly operate on temporaries and never access the register file.

Predication: Predication converts control dependences to data dependences. As described in later sections, predication enables the formation of large blocks essential for high concurrency. In Figure 3.1b, instead of using a conditional branch and a control dependent instruction to perform the increment, the compiler uses the test instruction `tlt` to compute a predicate and uses it to guard the execution of

the increment instruction—`addi_t`. The dotted arcs in Figure 3.1c depict these predicate dependences. The suffix (`_t` or `_f`) next to an instruction indicates that the instruction is predicated, and the value of the predicate at runtime determines whether the instruction will execute or not. The instruction must receive a predicate that matches its suffix in order to execute. For example, if the `tlt` instruction evaluates to false, the dependent `addi_t` instruction will not execute, but the `mov_f` instruction will execute. However, if the `tlt` instruction evaluates to true, the `mov_f` instruction will not execute, and the `addi_t` instruction will execute. The absence of a suffix indicates that the instruction is not predicated.

Target encoding: Instructions do not encode their source operands; they encode only their consumers. For example, in Figure 3.1d the encoding for the first load instruction specifies the consumer test instruction `tlt` using the label `N[5,R]`, which denotes that the result of the load is the right operand of the instruction labeled `N[5]`. Instruction `addi_t` produces a result that is live from the block—`num`. Hence it targets the write instruction denoted by `W[2]` that saves the result to the general register `$g2`. In the TRIPS architecture, the compiler assigns labels for all instructions in a block, and the hardware interprets these labels to map instructions to appropriate locations in the hardware.

Fanout: Encoding limitations prevent the ISA from specifying all of the consumers with the same producer instruction. In such cases, the ISA inserts additional `mov` instructions called *fanout* instructions to forward the results to every consumer. Figure 3.1d shows these fanout instructions. The example assumes that the encoding allows the specification of only one target with any instruction that consumes an immediate value. For example, the predicate computed by the instruction `tnei` is consumed by both the branch instructions, `bro_t` and `bro_f`. However only one of them can be encoded with the `tnei` instruction, resulting in an extra `mov` instruction,

labeled `N[12]`, that consumes the result from `tnei` and forwards it to the two branch instructions.

Block format: TRIPS blocks are single-entry, multiple-exit regions of instructions with no internal transfers of control. In TRIPS blocks, instructions do not contain any control dependences. The only dependences are true data dependences and dependences enforced via reads and writes to data memory. Every block must contain exactly one branch that will be taken at runtime. Furthermore, a taken branch must transfer control to a succeeding block, and not to another instruction within the same block. In our example, only one of the two branch instructions—`bro_t` or `bro_f`—will receive a matching predicate and alter the control flow in the program, leading it to either the beginning of the same block—`blk0`—or a different block—`blk1`.

Discussion

The TRIPS ISA specification is the culmination of much research that explored several design alternatives. In this section, we describe the progression of that research. That block-atomic execution had the potential to reduce per-instruction overheads was obvious in principle. To evaluate that premise, we examined several possibilities for constructing a block—basic blocks, traces [130], and hyperblocks [92]. Basic blocks are small and often contain fewer than six instructions. Amortizing the overheads of block-oriented execution requires larger blocks. Runtime traces are one solution, but they incur the hardware complexity of constructing a trace from dynamic distributed execution. Furthermore, they sacrifice the ability in the compiler to optimize for the latency of communication on a distributed substrate. Therefore, we settled on compiler-constructed hyperblocks, originally defined by Mahlke et al. as single-entry, multiple-exit regions of code with no internal control flow changes [92]. In an initial evaluation, we chose the hyperblocks produced by

Trimaran [4], which is a compilation framework targeted for VLIW/EPIC machines.

Our initial evaluation showed that hyperblocks are effective in statically combining several basic blocks from hot runtime control paths [134, 135]. Furthermore, large blocks exposed only few input and output register values for every block and confined most inter-instruction dependences within the boundaries of a single block [134]. Finally, nearly 80% of all instructions had fewer than two consumers, indicating low overheads for fanout [134]. Our subsequent research focused on defining the attributes of a block to reduce the complexity of a hardware implementation. The guiding design principles through that exploration were simple uniform mechanisms for two features—dataflow firing and inhibiting the effect of instructions that must not execute. To ease the hardware implementation, we added a few constraints to the block: fixed number of instructions, the maximum number of register read and write instructions, and the maximum number of load and store instructions. We discuss these constraints further in Chapter 4.

The delineation of register read and write instructions from other instructions helps in the quick identification of inter-block dependences. Prior to the complete execution of a block, the hardware can identify the set of registers defined in one block and resolve inter-block register dependences quickly. This identification of register definitions is similar to *create masks* used by Multiscalar processors [152]. The explicit read/write instructions also help preserve uniform dataflow firing rules for every instruction. Rather than requiring an instruction executing at a functional unit to fetch a required value from the register file, it allows the hardware to push values from the register file to the consumers and execute instructions in a dataflow fashion.

Directly encoding the targets with an instruction reduces overhead instructions required just for operand communication. Target encoding also reduces most of the operand communication to temporaries and minimizes the overall number of

accesses to the register file. An instruction can push values to its consumers and cause dataflow firing, instead of having the consumers read values from a centralized register file. However, as we described earlier, the presence of more targets than allowed by the encoding space mandate overhead instructions for fanout, degrading overall performance. The exact number of targets supported with each instruction is determined by the length of the encoding and whether the encoding supports fixed-length or variable-length formats. A fixed-length format simplifies the instruction decode logic, whereas a variable-length format reduces fanout instruction overhead. The TRIPS ISA chooses fixed-length formats to reduce hardware complexity.

Predication is a necessary artifact of large instruction blocks that do not support internal control flow changes. The lack of control flow changes within one block reduces the hardware overhead of distributed control flow synchronization that must communicate the taken/not-taken status of an internal branch to other instructions mapped on the distributed substrate [89]. Predication also eliminates hard-to-predict branches from the instruction flow, which helps uninterrupted instruction fetch and enlarges the parallelism scope for the compiler. However, retaining control dependences across blocks enables the hardware to use dynamic speculation and improve performance.

The hyperblocks originally generated by Trimaran featured multiple exits, of which exactly one will be taken at runtime. However, these exits complicate the precise identification of register definitions from a block. Early exits convey an implied sequentiality in the execution of instructions within the block, which is not a good match for dataflow execution. Instruction sequences separated by a taken branch may have no data dependences among them. Therefore inhibiting the dataflow execution of the instructions "following" the branch requires the communication of the branch status to those instructions. The TRIPS ISA takes the simpler approach of inhibiting instruction execution through the predication.

3.2.2 Distributed Microarchitecture

The TRIPS microarchitecture exploits the parallelism exposed by large blocks using wide issue and a large window of instructions. It is designed with the following principles: a) complete distribution, b) simple distributed components, and c) point-to-point communication. Distribution improves concurrency, whereas the use of simple components and point-to-point communication links improves scalability. Given these goals and benefits, numerous design possibilities exist for the microarchitecture. The overarching questions for a distributed organization are: a) what to distribute, b) how to distribute, and c) how to connect the distributed components. In addition, the microarchitecture must provide suitable mechanisms for conducting distributed execution—mapping instructions on to the distributed substrate, operand communication, atomic commit of architectural state, and speculation. This section describes the particular choices made in the TRIPS microarchitecture. In Section 3.4 we revisit these questions, discuss design alternatives, and the rationale behind our particular choices.

High Level Organization

Figure 3.2 shows the specific organization of the TRIPS prototype processor, which is the first hardware implementation of the TRIPS architecture [136]. It shows a tiled organization where the entire microarchitecture is partitioned into a number of tiles. Each tile implements a particular microarchitecture functionality, and the different tiles are connected together using simple point-to-point network links.

Figure 3.2 shows the different tiles in the microarchitecture and the functionality implemented by each. It shows a 4×4 array of execution tiles (ET) surrounded by register tiles (RT) along the top edge, one control tile (GT) at the top left corner, and instruction tiles (IT) and data tiles (DT) along the left edge. Each ET consists of a multi-entry reservation station that holds the operands and instruc-

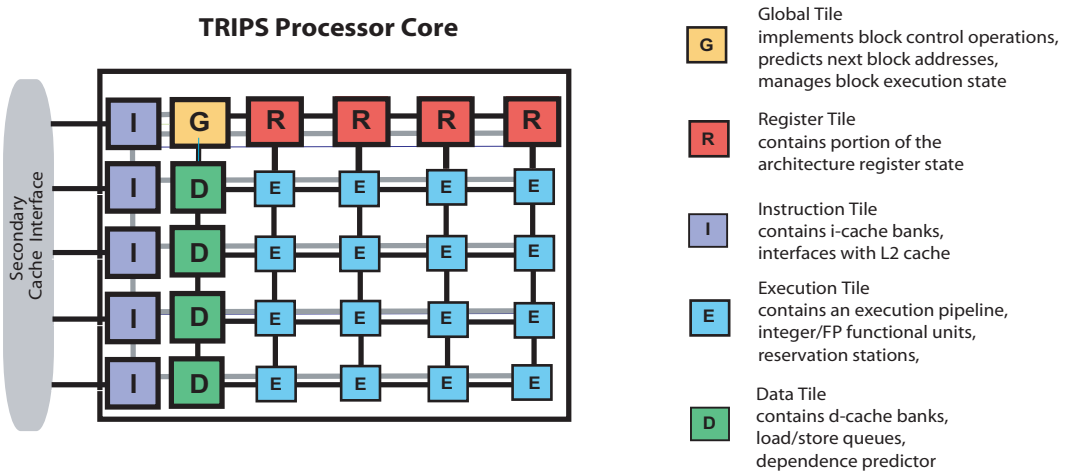


Figure 3.2: TRIPS processor microarchitecture organization. The microarchitecture consists of multiple, replicated, heterogeneous tiles connected with each other by point-to-point interconnection links.

tions destined for that tile, a standard execution pipeline that fires an instruction as soon as all of its operands are available, and router connections to its neighbors. After executing an instruction, an ET can forward the result to another reservation station slot in the same or different ET using the routing network.

The ITs and the DTs implement the distributed primary memory system for data and instructions respectively. Each IT couples an instruction cache bank with the row of ETs or RTs in the same row. The IT at the top-most row caches the read and write instructions, whereas the ITs in the rest of the rows cache the instructions for their corresponding row of ETs. Each DT contains a data cache bank and a load-store queue for sequencing the memory operations in program order. The DTs are address partitioned. The hardware executes a load or store instruction by computing the effective address at an ET and forwarding the computed address to one of the DTs. The targeted DT performs the required operation, and for a load, forwards the result directly to the consumer instruction using the routing network. The RTs at the top row statically partition the architecture register state among them and

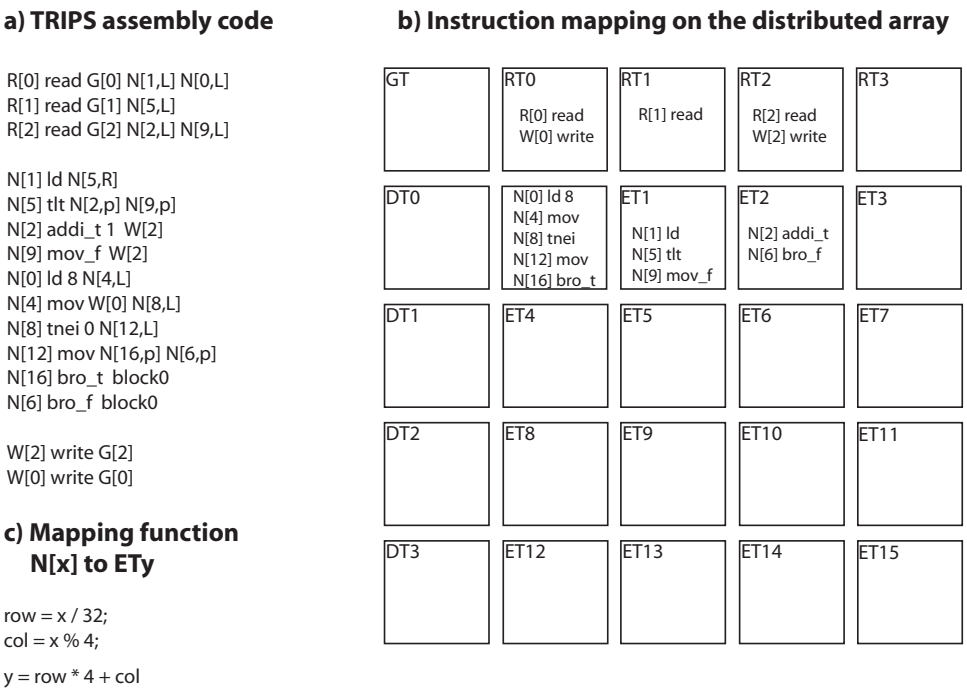


Figure 3.3: Mapping of instructions in a block to different tiles.

supply the register values to the ETs using the routing network. The singleton GT sequences and manages the overall execution, and is the only centralized resource in the microarchitecture.

The microarchitecture combines the reservation stations across all ETs to form a large, distributed instruction window, and executes instructions out-of-order using dataflow firing rules. It translates static instruction labels assigned by the compiler to physical locations in the window where the instructions must execute and where the operands must be transmitted. Figure 3.3 depicts the assignments of the instructions in our example block to physical tile locations. For example, the `addi_t` instruction that carries the label `N[2]` is assigned to `ET2`, and the `tlt` instruction that carries the label `N[5]` is assigned to `ET1`. Read and write instructions

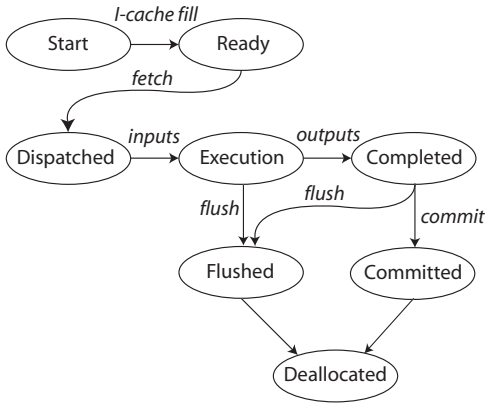


Figure 3.4: Different states in the execution of a block.

are assigned to the RTs as shown in the figure. Figure 3.3 shows the specific mapping function from the instruction labels to the execution tiles in the microarchitecture. Other implementations may choose different mappings.

As shown in Figure 3.3, the hardware essentially maps the instructions in a block to a 3-D volume of reservation stations. The x-dimension and the y-dimension determine the ET, and the z-dimension determines the reservation station slot within the ET where an instruction must be mapped. The compiler assigns labels seeking a hardware placement on to the 3-D topology that minimizes the physical distances that operands must travel along the block's critical dependence paths.

Block Execution

The distributed execution of each TRIPS block begins and ends at the GT. As shown in Figure 3.4, a block goes through different states during its execution in the microarchitecture. This section describes the various states and the transitions among them.

- *Start* — The life of a block begins when the GT determines that this block is the next one to execute on the processor.

- *Ready* — If the ITs do not contain the block, the GT instructs them to fetch the block from the secondary memory system. Subsequently, the block becomes resident in the I-cache. The GT then allocates the necessary resources and prepares the microarchitecture for executing the block.

- *Dispatched* — The GT instructs each IT to access their respective cache banks and dispatch the instructions. The ITs dispatch read and write instructions to the RTs and other instructions to the ETs.

- *Execution* — An instruction fires as soon as all of its operands are available. After execution, it sends the result to the targets specified in the instruction encoding using the routing network. Read instructions execute by reading the values from the register file and sending it to their consumer instructions. For example, using the instruction mapping shown in Figure 3.3, the read instruction `R[2]` fires by reading the register `$g2` and sending it to the `addi_t` instruction at `ET2`. The `addi_t` instruction fires upon receipt of the value and subsequently sends the computed result to the write instruction `W[2]` instruction. A load instruction computes the address at an ET and sends the address to one of the DTs. Since the DTs are address partitioned, the value of the address determines the particular DT to which the ET will send the address. The targeted DT performs the load operation subsequently and forwards the value to the consumers directly. For example, if the address computed by the load instruction `N[1]`—after it fires at `ET1`—is destined for `DT3`, then `DT3` will receive the address, perform the cache lookup, and send the load result to the consumer instruction `N[5]`.

- *Completed* — Instructions execute, operands trickle through the network, and eventually, the block produces outputs—branches, registers and stores, which are routed to the RTs and the DTs. After all outputs of the block have been

produced, the block is said to have completed its execution. For example, in Figure 3.3, the block completes its execution after `RT0` and `RT2` have both received the values for their respective write instructions and the GT has received the branch target from either `N[16]` or `N[6]`.

- *Flushed* — If the execution of the block must be annulled for any reason, the GT instructs all tiles to flush the execution state corresponding to the block.

- *Committed* — If the execution of a block completed without any exceptions, the GT directs the RTs and the DTs to commit the outputs produced by the block. The RTs and DTs independently commit the architectural state modified by the block.

- *Deallocated* — After a flush or a commit operation has completed, the GT reclaims the resources allocated for the block and reserves it for use by the next block.

The execution of a single block involves all the tiles in the microarchitecture. For example, the RTs are needed to provide register values, the DTs for memory operations, and the ETs for executing instructions within the block. The microarchitecture must implement precise protocols and networks to control the operations in each tile and manage the overall execution of a block. Chapter 4 presents the details of these protocols for the TRIPS prototype processor.

Executing Multiple Blocks

Although the previous sections described the block operations as indivisible, the hardware can exploit concurrency within and across various operations. For example, the dispatch of each instruction is independent of the others, which allows different ITs to dispatch instructions concurrently. Even as the ITs are dispatching instructions, a few ETs may begin to execute their instructions, if they have their

operands available. Aside from the intra-block concurrency, the hardware can use speculation to exploit concurrency across multiple blocks. This section describes the necessary hardware for support for speculative execution.

The reservation stations distributed throughout the array of ETs provide a window on to which the hardware can map instructions and exploit ILP. The hardware can expand the window by merely augmenting each ET with additional reservation station entries without the attendant complexity present in conventional superscalar machines. It can then use the additional slots to map multiple blocks speculatively and execute them. Different possibilities exist for partitioning the reservation stations among multiple blocks. Specific implementations may choose to devote all slots within the same row to the same block, and different rows to different blocks. Alternative implementations may devote all slots within the same column to the same block. These schemes offer different tradeoffs with respect to the latency of operand communication within the block. Section 3.4 discusses these tradeoffs in greater detail.

In the TRIPS microarchitecture, the hardware maps an entire block to all rows and columns of ETs. It groups a set of reservation station in each ET and aggregates them across all ETs to form a *frame*. Figure 3.5 shows the partitioning of the reservation stations at one ET into eight different frames, with each frame containing four slots per ET. If each frame supported eight frames per ET, the 4×4 array of ETs provides 128 slots per frame on to which the hardware can map an entire 128-instruction block. A multi-entry reservation station at each ET thus provides a three-dimensional volume of instruction storage across the entire array as shown in Figure 3.5. On to this volume, the hardware can map multiple blocks and execute them concurrently.

The hardware fills empty frames with speculatively mapped blocks, predicting which blocks to execute next and mapping them to empty frames. The hardware

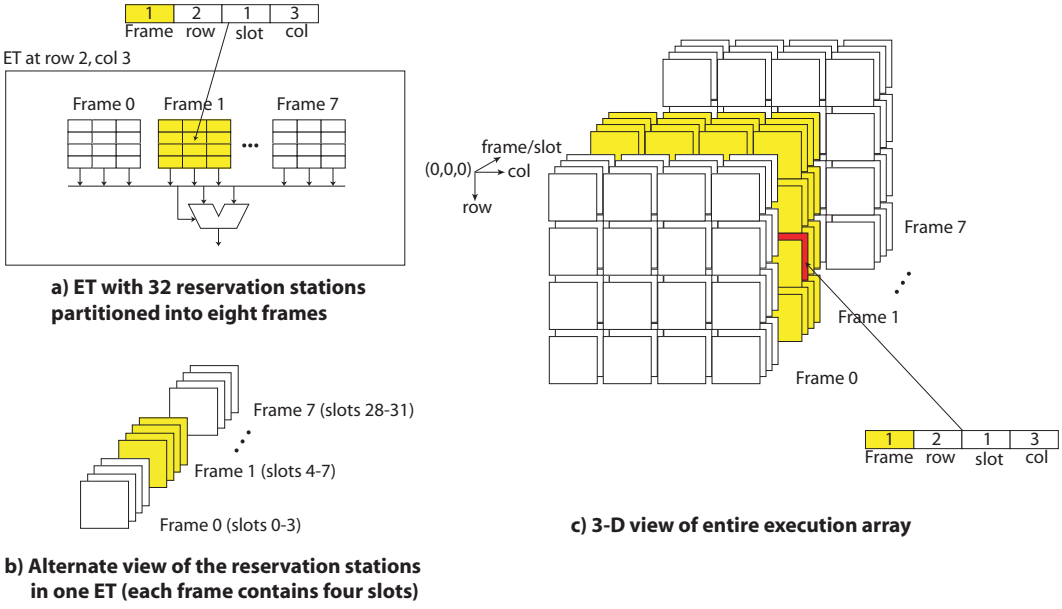


Figure 3.5: Organization of reservation station slots into different frames.

uses a control flow predictor to predict the address of the next block that must be executed in the program. It can map the block to an available frame and execute it, even as the previous one is still executing. The frames are treated as a circular buffer in which the oldest frame is non-speculative and all other frames are speculative. When the frame holding the oldest block completes, the block is committed and removed. The next oldest block becomes non-speculative, and the released frame can be filled with a new speculative block. On a misprediction, all blocks past the incorrect prediction are squashed and restarted.

Since frames are assigned to blocks dynamically and all intra-block communication occurs within a single frame, each producer instruction prepends its frame identifier to the 3-D coordinates of the consumer to form its correct reservation station address. Figure 3.5 shows an example of how this addressing works. The row and the column components of the coordinates determine the ET, and the frame

identifier together with the slot component of the coordinates determines the reservation station address. Data passed between blocks are transmitted through the register file or the load-store queues. Register values are forwarded aggressively when they are produced, using forwarding logic at the RTs that match the outputs of earlier blocks with the inputs of later blocks. The precise identification of the register definitions within the block enables the hardware to identify inter-block dependences quickly. The load-store queues use a similar mechanism to forward values from stores to consuming loads in other blocks.

3.2.3 Discussion

The TRIPS microarchitecture exhibits several advantages over conventional superscalar architectures for exploiting concurrency.

Instruction fetch: A single prediction is sufficient for fetching an entire block of instructions. The next-block predictor provides a single block address, and the ITs independently fetch and dispatch instructions for the block, thus providing a highly parallel instruction fetch interface. Furthermore, the hardware reduces the pressure on the next-block predictor to make fast predictions, as the fetch of a large block may incur multiple cycles, offering sufficient slack for making an accurate prediction for the next block. By contrast, modern control flow predictors must often trade off accuracy with prediction latency in order to sustain high-bandwidth instruction fetch [78].

Data memory: The distributed data cache banks offer a high-bandwidth primary memory system interface. Since the DTs are address-partitioned, they can support multiple independent accesses simultaneously. However, maintaining correct program-order semantics for loads and stores efficiently is an important design issue for distributed primary memory interfaces and is the subject of other work [141].

Register interface: Large blocks reduce many data dependences to strictly intra-block, direct-instruction communication. They alleviate the register pressure as most dependences do not need to access the register file. An initial evaluation indicated that by converting most dependences to intra-block dependences, 30%–90% reduction in register bandwidth be achieved [109]. Furthermore, for the residual register accesses that are necessary for inter-block dependences, the hardware provides distributed register banks, offering concurrent accesses to different registers. Across blocks, registers must still be renamed to eliminate anti- and output-data hazards, but the few block input and output registers reduce the complexity of the rename logic.

Operand communication: The hardware connects the distributed components with point-to-point interconnects, instead of an unscalable bypass network. The compiler optimizes for the latency of operand communication through the critical path of dependent instructions.

Wide issue: The array of execution units provide both wide issue and a large window of instructions. Parallelism results from concurrent execution in different ETs, restricted only by the data dependences. The explicit expression of dependences obviate associative operand tag match hardware, which improves the scalability to wider issue.

3.3 Compiling for TRIPS

The compiler for the TRIPS architecture has many of the same responsibilities as a classical optimizing compiler. In addition, it has two new responsibilities: a) decompose a program into a sequence of blocks and b) perform instruction scheduling. This section describes the compiler transformations that accomplish these tasks.

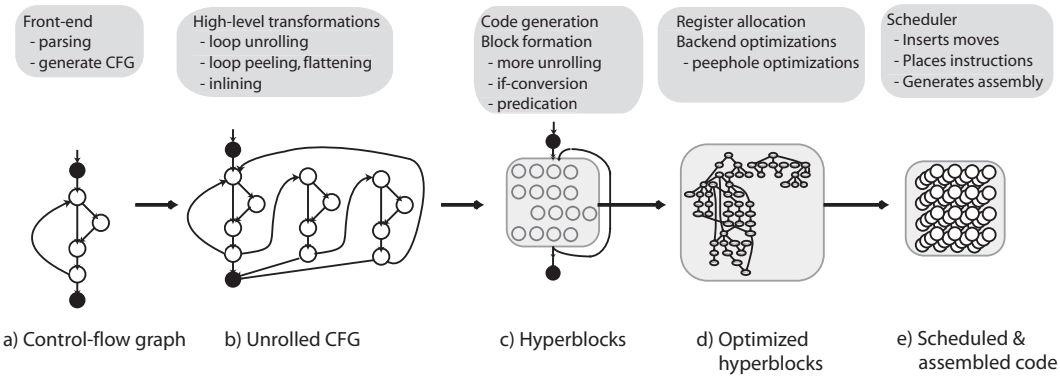


Figure 3.6: TRIPS compiler phases.

3.3.1 Scale Framework

The TRIPS compiler is based on Scale, a Java-based compilation framework originally targeted for Alpha and SPARC architectures [3]. Figure 3.6 depicts the typical phases in the TRIPS compiler. The compiler uses a front-end to parse source programs written in C and FORTRAN and transforms them into control flow graphs. The first part of the figure shows a typical control flow graph (CFG), where each node represents a single basic block. The compiler then performs several high-level transformations such as loop unrolling and flattening to expand the CFGs. It then performs several scalar optimizations to produce efficient code: global variable replacement, sparse conditional constant propagation, copy propagation, loop invariant code motion, useless copy removal, dead code elimination and dependence analysis. Figure 3.6b shows the CFG after these transformations. The compiler then generates TRIPS instructions and forms TRIPS blocks using additional unrolling, if-conversion, and tail duplication. It then performs various predication optimizations, register allocation, and peephole optimizations. It finally schedules instructions and assembles the final object code. Figures 3.6c–e depict these phases.

3.3.2 Hyperblock Formation

Hyperblock formation is a transformation that combines several basic blocks, including those on disjoint control paths, into a larger block called the hyperblock [92]. To expose opportunities for hyperblock formation, Scale uses transformations such as function inlining, loop unrolling, loop peeling, loop flattening, if-conversion and tail duplication [91]. All of these optimizations, besides exposing opportunities for scalar optimizations, eliminate control boundaries that inhibit hyperblock formation. Scale then combines several basic blocks to form a larger hyperblock. It uses predication to merge basic blocks from disjoint control paths into the same hyperblock. To maximize the chances of merging only useful basic blocks into a hyperblock, Scale can use heuristics based on profile information such as basic block execution frequencies, control flow edge frequencies, sizes of the basic blocks, and loop counts.

Figure 3.6 shows these transformations of the while-loop code example depicted earlier in this section. The first portion shows the CFG of the loop. Each circle in the graph represents a single basic block and the solid circles mark the prologues and epilogues for the loop. The compiler then unrolls the loop thrice as shown in the second portion of the figure. It decides to merge all 12 basic blocks corresponding to the three loop iterations into the same hyperblock. It performs if-conversion and predication and forms one large hyperblock as shown in Figure 3.6c. It then performs several backend optimizations, and does register allocation. Finally it performs instruction scheduling and produces TRIPS object code.

3.3.3 Predication

Predication is an architectural concept in which the execution of an instruction is guarded by a predicate operand [8]. A compiler transformation called if-conversion converts control dependences to data dependences, by computing a predicate and

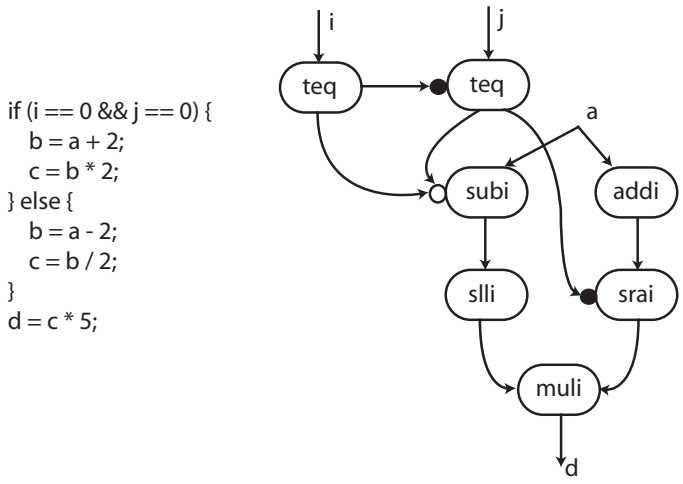


Figure 3.7: Predication in the TRIPS architecture.

making the control dependent instructions data dependent on that predicate. It is this transformation that enables basic blocks on disjoint control paths to be part of the same hyperblock. The TRIPS architecture exploits dataflow execution to implement predication efficiently [149].

Nearly all instructions can be predicated in the TRIPS architecture. A special field with every instruction specifies whether an instruction is predicated and if so, the polarity of the predicate on which the instruction must be executed. Figure 3.7 illustrates the various features of predicated execution in the TRIPS architecture—*predicate ANDs*, *predicate ORs*, and *implicit predication*. We provide an overview of these features in the following paragraphs and refer the reader to prior work for a comprehensive treatment on the subject [149].

The figure shows an if-then-else construct with a short-circuiting conditional computation. Two test instructions, the second predicated upon the first, compute the conditions for executing either the if statement or the else statement. The circle next to an instruction specifies the polarity of the predicate on which the instruction will execute—a black circle indicates a predicate value of true, while a white circle

indicates a predicate value of false. For example, if the first test instruction `teq` evaluates to false, the `subi` instruction will receive a matching predicate and will execute when it has its data operand available. The second `teq` instruction will not execute, since it will receive a non-matching predicate.

Predicate ANDs: The execution of the if statement must be protected by a compound predicate that is the boolean conjunction of the two test instructions. By predicating the second `teq` instruction on the true condition of the first `teq` instruction, the compiler implements an implicit *AND* chain. This implementation, enabled by predicated test instructions, is more efficient than the explicit conjunction operators, which increase both the code size and the critical path length.

Predicate ORs: The execution of the else statement must be triggered if either of the test instructions evaluate to false. Instead of an explicit disjunction of the outputs from the two test instructions, the TRIPS ISA supports an implicit predicate-OR operation. If the first `teq` instruction evaluates to true and the second evaluates to false, the `subi` instruction receives two values for the predicate operand, but only one of them is a matching predicate. If the first `teq` instruction evaluates to false, the second `teq` instruction does not execute as it receives a non-matching predicate. Therefore, regardless of the outcome of the two test instructions, the `subi` instruction will receive at most one matching predicate. By targeting multiple predicates that are defined on mutually exclusive paths to the same predicate operand, the TRIPS compiler implements an implicit predicate-OR operation. As we describe in Chapter 5, this feature exposes more opportunities for optimization.

Implicit predication: Prior predication architectures required every predicated instruction to read its predicate operand explicitly. The same requirement for the TRIPS architecture would incur significant overhead. For example, if the execution

of a basic block is guarded by the predicate `p`, then the single predicate must be delivered to every instruction in the basic block resulting in increased fanout overhead. Fortunately, the TRIPS compiler can exploit the dataflow execution to mitigate this overhead. It uses two techniques: hoisting and implicit predication. For example, on the right-hand side of the if-then-else fork, the compiler predicates only the bottom `srai` instruction on the true value of the predicate, effectively hoisting the dataflow antecedent to execute speculatively in parallel with the predicate computation. If the predicate for the bottom instruction evaluates to false, the effect of the hoisted instruction is automatically nullified. On the left-hand side of the if-then-else fork, the compiler predicates only the top instruction, implicitly predicating its dataflow descendant. If the predicate for the `subi` instruction is non-matching, it will not fire, so the implicitly predicated dataflow chain also will not fire.

After all of the predication transformations, the TRIPS compiler produces a TRIPS block that includes instructions from several basic blocks, and among which, the only instruction dependences are data dependences. Its next step is to express the inter-block register dependences using architecture register names.

3.3.4 Register Allocation

Recall from the TIL code example in Figure 3.1 that register names express data dependences between instructions, unless the dependences must be expressed through data memory. Register allocation assigns these names to architecture registers or spills to memory. However, unlike traditional architectures, the register allocator does not assign registers whose live ranges are contained entirely within the bounds of a single block. For the purposes of register allocation, the compiler treats each block as a large instruction that uses and defines several registers. It prioritizes the registers based on their definitions, uses, and spill costs, and also the size of the block that uses and defines them. It then performs a partitioned register assignment

or spills to memory in priority order. Specific constraints in the ISA—such as the maximum number of definitions allowed in a single RT—may cause the allocator to iterate several times until all constraints are satisfied [147].

3.3.5 Instruction Scheduling

The final phase of compilation is instruction scheduling. The TRIPS architecture breaks instruction scheduling into two complementary components: *instruction placement* and *instruction issue* [108]. Conventional architectures sit at opposite ends of the spectrum with regard to these demands on the scheduler. VLIW processors use both static placement (SP) and static issue (SI), resulting in a SPSI approach to scheduling. Out-of-order superscalar architectures, conversely, rely on both dynamic placement (DP), since instructions are dynamically assigned to appropriate ALUs, and dynamic issue (DI), resulting in a DPDI model. For VLIW processors, the static issue is the limiting factor for high ILP, whereas for superscalar processors, poor dynamic placement locality limits ILP. Static placement makes VLIW a good match for partitioned architectures, and dynamic issue permits superscalar processors to exploit parallelism and tolerate uncertain latencies. The TRIPS architecture combines the strengths of these two models, coupling static placement with dynamic issue, resulting in an SPDI model. The TRIPS compiler optimizes for placement on the hardware, and the hardware issues instructions dynamically, as their operands become available.

The TRIPS scheduler has the following responsibilities: *a) Placement for Locality:* select an instruction mapping that minimizes communication latencies among execution resources, the register file, and cache banks for dependent operations and *b) Contention Reduction:* reduce contention by spreading independent instructions across the execution resources, balancing this benefit against the goal of reducing communication latencies. The TRIPS scheduler uses an algorithm called Spatial

Path Scheduling to perform these two functions [38]. It is an improvement over the previously proposed greedy list scheduling algorithm [108] and exploits the following techniques:

Anchor points: Exploits known routing locations, which includes register destinations, branch destinations, approximate memory instruction destinations, and previously scheduled instructions to determine the best placement for an instruction.

Contention modeling: Estimates contention for an ALU—both intra-block and inter-block—and the network links, and augments them with static properties of a block such as its size to balance the load across all ALUs.

Global register prioritization: Prioritizes for critical global paths that transcend block boundaries by considering loop-carried register dependences and register dependences across neighboring blocks.

Path volume scheduling: Plans routes for long paths by considering an entire group of instructions for placement, instead of a single instruction at a time.

Fanout generation: Estimates criticality of various consumers and constructs necessary fanout trees from producer instructions.

The TRIPS scheduler repeats this operation for every block in the program. Finally, the TRIPS assembler encodes the instructions in every block according to the ISA and produces the final object code that will execute on the hardware.

3.4 Design Alternatives

In Section 3.2, we described the features of the TRIPS microarchitecture. Three important issues arise when designing a distributed microarchitecture such as TRIPS:

a) what to distribute, b) how to distribute, and c) how to connect the distributed components. In addition to the exploration of the ISA design space, our initial research also focused on exploring the design space of the microarchitecture. In this section, we describe a few design alternatives and the rationale behind the design parameters chosen by the TRIPS microarchitecture.

3.4.1 What to Distribute

The first issue with distribution is deciding which microarchitectural components to distribute. The TRIPS microarchitecture targets the centralized structures that are fundamental impediments for scalability in conventional superscalar processors. It replaces the centralized primary memory system with distributed instruction cache banks, distributed data cache banks, and distributed LSQs. It replaces the centralized register file and register rename tables with distributed register file banks. It replaces the centralized OOO instruction scheduler with multiple simple dataflow schedulers distributed throughout the array of execution units. Finally, it replaces the broadcast operand bypass buses with a point-to-point interconnection network.

The only centralized component is the GT, which includes the next-block predictor and the block control logic. Fortunately, both of these structures are accessed only at the block boundaries and not for every instruction. By using the large blocks, the microarchitecture amortizes the overhead of directing the overall execution from a single centralized component. A completely distributed implementation for control flow management is beyond the scope of this dissertation and is the subject of other work [125].

3.4.2 How to Distribute

The second issue deals with the number of partitions for each distributed component and how they must be organized. The fundamental tradeoff that decides

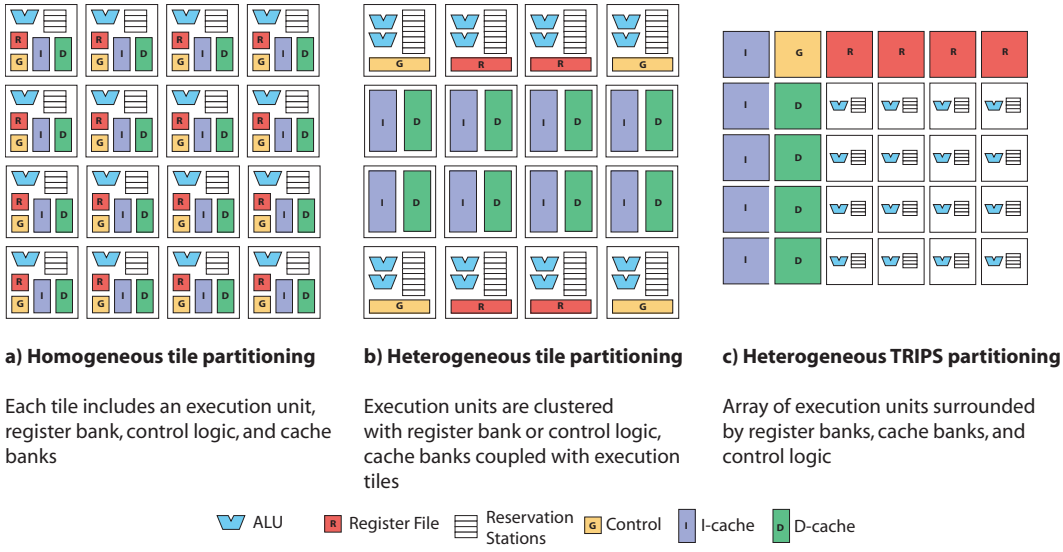


Figure 3.8: Different organizations of the distributed components.

the distributed organization is the complexity in each tile versus the communication latency between the tiles. Figure 3.8 depicts a few organizations of a distributed microarchitecture. Each organization depicts a different arrangement of the individual components. For example, Figure 3.8a shows a 4×4 array of tiles, with each tile consisting of an execution unit, register file, reservation stations, caches, and a control predictor. This organization has equal number of partitions for each distributed component and yields a homogeneous arrangement in which each tile integrates several distinct components, and multiple identical tiles form the microarchitecture. Figure 3.8c depicts the heterogeneous organization of the TRIPS microarchitecture. It features unequal number of partitions for the different components and multiple replicated heterogeneous components comprise the microarchitecture. Figure 3.8b also shows a heterogeneous microarchitecture, but with a different organization.

The co-location of instructions and data with the execution unit in the homogeneous organization reduces the communication latency for loads, stores, and

instructions that access the register file, and also improves the bandwidth. However, it increases the area of each tile, or conversely, forces a reduction in the capacities of the individual components. Larger tiles increase the inter-tile communication latencies and also expose wire delays within one tile. A homogeneous organization must also deal with other design issues:

Instruction cache: The instruction cache is the simplest to distribute and co-locate with the execution units. If instruction mappings to execution units do not change dynamically, or change infrequently, this organization reduces the latency of dispatching instructions to the execution units and improves instruction fetch bandwidth, but increase tile area.

Register file: Replicating the registers across different register banks requires mechanisms to maintain consistent data for the same register in multiple banks. Partitioning the registers statically among different register banks increases the constraints for static register allocation; the instruction scheduler must work in conjunction with the register allocator to minimize remote accesses for register data, increasing its constraints.

Data memory: A distributed primary memory system must deal with two important issues—cache coherency and load/store synchronization. Static partitioning of the address space among the tiles obviates the need for explicit cache coherence. But this policy requires sophisticated data partitioning and instruction scheduling algorithms to minimize the occurrences of a remote cache bank access. Alternative designs may choose replication, instead of hard partitioning, and migrate data closer to tiles that access them. But, they must incur the overhead of keeping the caches coherent.

Efficient synchronization mechanisms are also necessary for enforcing the

correct program order of loads and stores. Since addresses for loads and stores cannot be perfectly disambiguated at compile time, a single partition must provide runtime mechanisms for servicing all possible in-flight memory operations. Naive policies incur a complexity that is proportional to the number of partitions and therefore, do not scale. Sethumadhavan et al. describe these issues in greater detail in recent work [140, 141].

When we began the implementation of the TRIPS microarchitecture, efficient policies for register and data distribution were still being investigated. Consequently, the TRIPS microarchitecture opted for the simpler middle ground of fewer partitions of the register file, the data caches, and the LSQs. This policy offers the benefits of higher bandwidth and scalability than a centralized mechanism, but without the additional complexity of more distribution. Furthermore, to keep the tiles simple, the microarchitecture pushed the register and cache banks to the edges of the execution array and relied on the compiler to reduce communication latencies. Using large blocks generated by the compiler, the hardware also expected to amortize the latencies of traversing to the edges of the array for accessing registers and caches.

3.4.3 How to Connect

The third issue for distributed architectures is the interconnection network that connects the different tiles. Taylor et al. [164] and Sankaralingam et al. [137] describe different taxonomies for on-chip interconnection networks, and in particular, operand networks that connect the partitions in a tiled architecture [164]. Routed on-chip networks are also popular as an alternative to global interconnects to reduce wire delays [41]. The fundamental design issues for these networks include the topology of the network, routing protocols, scalability to larger topologies, and deadlock detection and avoidance. There is a vast design space for each and their treatment is beyond the scope of this dissertation. In the following paragraphs, we

highlight the basic design choices we made and their rationale.

The topology determines the latency and bandwidth of the network, which in turn affect performance. During the early design space evaluation of the TRIPS architecture, we considered a low-degree M-network, which connects each tile to the three nearest neighbors (two diagonal, one below) in the succeeding row, to reduce the routing latency per-hop and approximately match the shape of the program DFGs [109]. Ultimately, the best topology is the one that provides sufficient reachability for any pair of nodes to communicate with each other, attains a good balance between the number of routing hops between communicating pairs and the latency per hop, enables simple routing algorithms, and matches well with the chip floorplan. The mesh network provided the best compromise. Its routers have four input and four output ports each, where as the routers in a M-network have only three input ports and three output ports. On the one hand, the additional ports increase the latency per hop in the mesh network. On the other hand, the mesh network yields fewer hops on average between any pair of tiles compared to the M-network. Singh et al. describe a study that explored different routing topologies and concluded that a simple mesh network attains much of the performance of a higher-connectivity star network, and superior performance compared to a low-connectivity M-network [146].

The TRIPS microarchitecture chooses a simple dimension-ordered dynamic routing protocol. The MIT RAW processor uses a statically scheduled network to optimize around congested links and avoid contention. However, RAW also needed a slow dynamic network to cope with unexpected events such as cache misses [163], which inhibit parallelism. We opted for dynamic routing to enable out-of-order execution. We also chose dimension-ordered routing and guaranteed delivery of packets to simplify the router design and avoid deadlocks. However, as our results in Chapter 6 illustrate, this policy leads to contention stalls under heavy traffic and

degrades performance.

3.4.4 Design Parameters

A number of design parameters exist for the TRIPS microarchitecture. Among these are the dimensions of the array, composition of the ETs, speculation depth, and mapping of the blocks.

Array Dimensions

The TRIPS prototype microarchitecture uses a 4×4 array of ETs. This organization offers a peak execution rate of 16 instructions per cycle. To increase the peak execution rate, alternate designs may increase the dimensions of the execution array. Sankaralingam et al. evaluate the scalability of ILP to larger cores [135]. They explore different array configurations—2×2, 4×4, 8×4, 8×8. A fixed organization of register banks and data cache banks at the edges of the execution array increases the latency for register access and load/store instructions in larger arrays, if the consuming instructions are placed at farther ETs. The increased latency trades off with the increased concurrency offered by larger arrays.

Sankaralingam et al. measure the performance of several SPEC CPU2000 workloads and conclude that the wide variance in ILP in the workloads demand both larger processors (8×8) and smaller processors (4×4) [135]. However, larger cores provide better overall performance for several workloads, with the 8×8 configuration performing the best. The 8×4 configuration attains nearly the same performance, whereas the 4×4 configuration reduces performance by 12% on integer workloads and nearly 50% on numeric workloads. Ultimately, the number of execution units that can be accommodated on the chip will be limited by available die area. Whereas the TRIPS prototype processor uses a 4×4 configuration, larger arrays offer a scalability path for increasing performance at future technolo-

gies. Workloads with abundant ILP will continue to exploit the increased execution bandwidth. However, the microarchitecture must introduce new mechanisms to improve the efficiency of execution in workloads with low available ILP.

Speculation Depth

The number of frames in the microarchitecture determines the instruction window size. The microarchitecture can map several blocks in the available frames speculatively, thus providing a larger window for exploiting ILP. For example, a microarchitecture with 8 available frames, where each frame can accommodate a single 128-instruction block, provides a window of up to 1024 instructions. The accuracy of control speculation determines the effective utilization of the frames. The available die area also restricts the number of frames, as each tile must provide support for more speculative blocks and the increased number of in-flight instructions. Due, in a large part, to the area constraints and branch misprediction rate, the TRIPS prototype processor supports only eight frames.

Block Mapping

The TRIPS microarchitecture maps a single block across the reservation stations in all ETs. Alternate mapping policies may choose just a subset of ETs to map a single block. Figure 3.9 depicts the mapping of two blocks using three different policies. In the first policy, *Uniform*, the microarchitecture distributes the contents of each block across all ETs. In the second policy, *Vertical*, the microarchitecture chooses a subset of the ETs to map a single block. The figure shows a mapping, in which the first block, `block0`, is mapped to the left half of the ET array, whereas the second block, `block1` is mapped to the right half. The third policy, *Horizontal*, also chooses a subset of the ETs to map a single block. However, instead of partitioning the scheduling volume vertically, it partitions the volume horizontally for the two

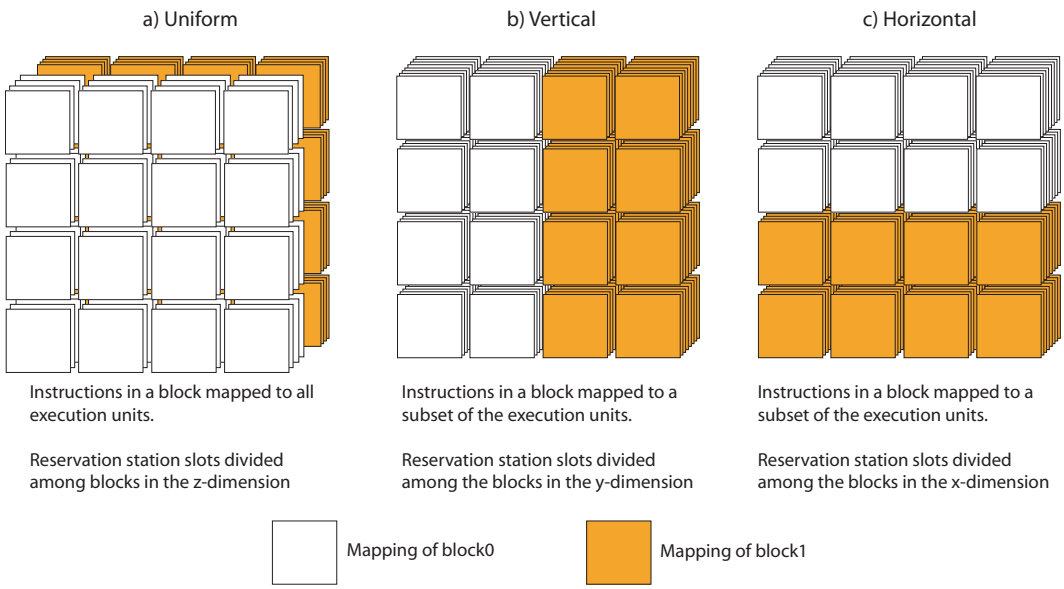


Figure 3.9: Mapping of different blocks to the reservation stations.

blocks.

The two mapping policies—*Uniform* and *Vertical*—represent opposite ends of the spectrum with respect to exploiting parallelism and minimizing communication. *Uniform* chooses to maximize parallelism among the instructions of the same block, whereas the *Vertical* chooses to maximize parallelism among multiple blocks. *Uniform* optimizes for communication with register files and data caches, whereas *Vertical* optimizes for communication among the ETs for a single block. For example, the maximum latency for intra-block communication among the ETs is 6 cycles for *Uniform*, while it is 4 cycles for *Vertical*, assuming single-cycle hops in the network. On the other hand, the minimum roundtrip load-to-use latency for loads in `block1` is six cycles for *Vertical*, as opposed to two cycles for *Uniform*. In fact, the fixed organization of the register banks and data cache banks hamstrings every mapping policy other than *Uniform* with respect to load and register communication. We chose *Uniform* to optimize for the loads and registers.

3.5 Summary

In this chapter, we described the architectural and microarchitectural principles of a TRIPS processor. We began by describing EDGE architectures, a new class of instruction set architectures that convey instruction dependences explicitly to the hardware. We described how these architectures elevate the granularity of processing to large program regions called blocks and reduce the instruction-level overheads of register file access, branch prediction, renaming, and dynamic scheduling that are present in conventional superscalar architectures. We then described the TRIPS architecture, an instance of an EDGE architecture, which executes instructions in dynamic dataflow order on a distributed array of heterogeneous processing tiles. We described how the architecture uses compiler assistance to reduce the latency of distributed execution, resulting in a statically placed (SP) and dynamically issuing (DI) variant of an EDGE architecture. We described the microarchitectural execution of a program on a particular implementation of the TRIPS architecture.

We described various design alternatives and discussed their relative merits. The choices made in the TRIPS architecture and microarchitecture represent only one point in the design space. Few of the alternatives such as a homogeneous organization and different block mapping policies show promise and merit further exploration. As our results in later chapters show, operand communication latency is a significant determinant of performance, and these alternatives present different tradeoffs for reducing latency. However, other decisions such as the fixed static mapping of instructions to execution units and the mesh network topology are likely best matches for distributed microarchitectures. Static mapping offers the ability to exploit the compiler to reduce latency. A mesh network provides simple routing algorithms, good performance, and maps well with the straight-line, physical wiring tracks in the hardware. Different implementations of the TRIPS architecture are likely to retain these two design choices.

Chapter 4

The TRIPS Prototype Implementation

Previous chapters described the basic principles behind the TRIPS architecture. A high-level exploration quantified the merits of the architecture and demonstrated its ability to exploit significant instruction-level parallelism [109]. However, that study omitted several low-level implementation details, the design challenges that were involved, and the potential performance overheads of a hardware implementation. The development of the TRIPS hardware prototype system is a comprehensive effort to understand those issues [136]. This chapter describes the details of the prototype implementation. In subsequent chapters, we use the prototype implementation for a detailed evaluation of the TRIPS architecture.

The TRIPS prototype chip is a single-chip multiprocessor consisting of two 16-wide TRIPS processors and a shared 1MB NUCA L2 cache [136]. It is implemented in a 130 nm IBM ASIC process and consists of more than 170 million transistors on a die area of 336 mm^2. Each processor itself implements the TRIPS architecture. This chapter presents the salient features of the prototype ISA, microarchitecture, and the chip implementation in Section 4.1 and Section 4.2. Sankar-

alingam also describes similar details in his dissertation [132].

This chapter then describes two aspects of the TRIPS prototype implementation that are the direct contributions of this dissertation: a) implementation of the global protocols and the control logic that manages the overall execution in the TRIPS processor, and b) performance validation of the prototype processor. Section 4.4 and Section 4.5 present these details. We use an understanding of the mechanisms described in these sections to drive the methodology for a detailed quantitative evaluation, which is the subject in later chapters.

4.1 The TRIPS Prototype ISA

The TRIPS processor is a hardware implementation of the TRIPS architecture. For ease of implementation, the prototype ISA places several restrictions on the composition of a block. First, each block must obey a few control flow restrictions; it may not have any internal transfers of control and may execute only one branch. In addition, each block may have only a maximum of 128 instructions, of which no more than 32 can be load or store instructions, and read up to 32 registers and write up to 32 registers. Furthermore, each execution of the block must emit exactly the same number of register outputs and stores. As we describe in Section 4.4, this restriction enables the hardware to identify the expected number of outputs from the block using static information, simplifying the implementation of the block commit protocol. To ensure this restriction, the compiler must predicate multiple instructions that define the same register such that only one of them will execute at runtime. Moreover, if a register is conditionally defined on a predicated path, *null writes* must be defined on the complimentary paths to ensure that an output is always produced—a true output or a null output.

A null write to a register indicates that the register is not defined by the block. Similarly, null stores indicate that the corresponding stores will not be produced

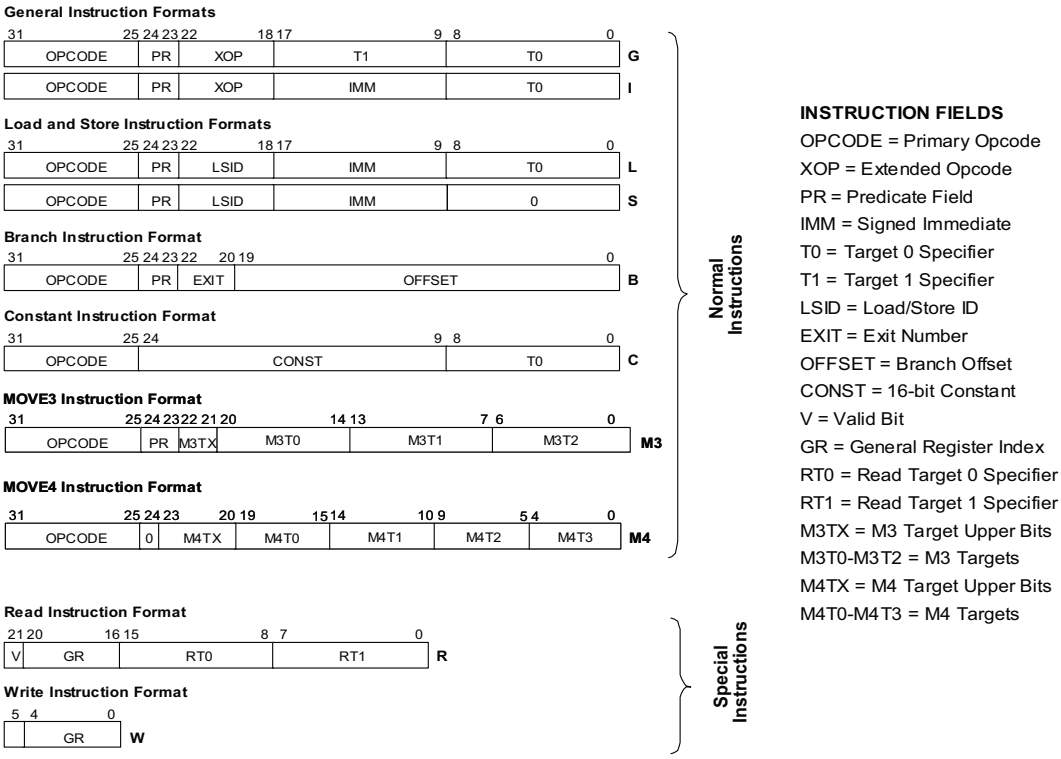


Figure 4.1: Instruction formats in the TRIPS prototype ISA.

when the block completes execution. The compiler inserts these null instructions such that whenever their complementary register defining or store instructions do not execute, the null instructions will execute and emit the outputs. For example, in Figure 3.1c, the compiler uses a `mov` instruction to copy the unmodified value of register `$g3` back to `$g3` in order to preserve its value whenever the if condition evaluates to false. Instead, the compiler could attain the same effect by replacing the `mov` instruction with a `null` instruction. The null instruction avoids the explicit read of the register and eliminates that dependence edge from the DFG, offering the potential for improved performance.

Table 4.1 presents a summary of the instructions in the ISA. The ISA provides instructions for accessing the registers, memory, integer and floating point

Category	Description	# instructions
Register access	Retrieves or writes to a general register.	2
Load	Retrieves a byte, half-word (2 bytes), word (4 bytes) or a double-word (8 bytes) from memory. Accesses must be aligned, and different instructions are provided for accessing signed and unsigned data.	7
Store	Modifies a byte, half-word, word or a double-word in memory.	4
Integer arithmetic	Supports integer arithmetic operations—add, subtract, multiply, divide. Supports mostly unsigned operations and immediate operands.	10
Integer logical	Supports bitwise logical operations—and, or, and xor. Supports immediate operands.	6
Integer shift	Supports signed and unsigned integer shift operations. Supports immediate operands.	6
Integer extend	Sign-extends to a 64-bit integer.	6
Integer comparison	Performs relational and equivalence tests on signed and unsigned integers. Supports immediate operands.	20
Floating-point arithmetic	Supports double-precision floating-point operations—add, subtract, multiply, and divide.	4
Floating-point conversion	Converts to or from single-precision and double-precision to integer.	4
Floating-Point comparison	Performs relational and equivalence tests.	6
Branch	Implements control flow.	6
Other	Generates large constants, move data values, and performs miscellaneous operations.	11
Total		92

Table 4.1: Summary of the instructions in the TRIPS prototype ISA.

arithmetic operations, logical operations, and relational operations. We refer the reader to the TRIPS ISA manual for a detailed description of these instructions [95]. The ISA encodes all instructions using exactly 32 bits. Figure 4.1 presents the encoding for various instruction formats. This encoding permits the specification of up to two targets in most instructions. However, instructions that consume immediate operands (I-form) and load instructions (L-form) can specify only one target. Additional targets, if any, must be specified by constructing a fanout tree of `mov` instructions. To minimize the number of fanout instructions, the ISA introduces two

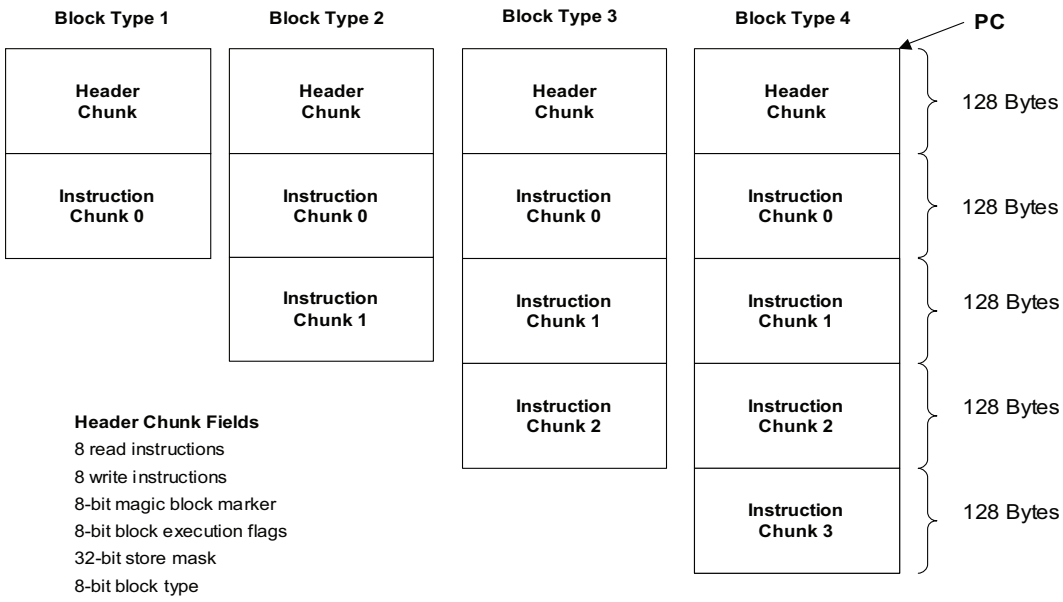


Figure 4.2: Block organization in the TRIPS prototype ISA.

special instructions—`mov3` and `mov4`—that specify three targets and four targets respectively. Figure 4.1 specifies the encoding formats for these instructions. Unlike other instructions that specify the complete coordinates for a target—row, column, reservation slot index, and the left/right/predicate operation position, the `mov3` and `mov4` instructions place restrictions on the locations of their targets. A `mov3` instruction requires all of its targets to address the same operand position—left, right, or predicate—specified by the `M3TX` bits in Figure 4.1. A `mov4` instruction requires all of its targets to target the same operand position *and* reservation station slots in the same row, as specified by the `M4TX` fields in Figure 4.1. These two instructions sacrifice the generality of the target encoding to reduce the overhead of fanout.

The prototype ISA arranges the instructions of a block in up to five 128-byte chunks. Figure 4.2 depicts the organization for four types of blocks. The header chunk encodes all the read and write instructions. It also encodes meta information about the block: a magic marker that identifies a legal block, the number of instruc-

tions chunks, control flags that specify any special execution modes for the block, and a 32-bit mask that indicates which of the 32 memory operations in the block are stores. Each instruction chunk encodes up to 32 compute instructions. Every block must include a header chunk and instruction chunk 0, and may optionally include the remaining instruction chunks. If an instruction chunk has fewer than 32 compute instructions, it is padded with NOP instructions to compose a fixed-size chunk. While the NOPs are not a requirement for the TRIPS architecture or EDGE architectures, the prototype ISA uses them for managing implementation complexity. We will revisit the benefit of fixed size chunks later in Section 4.4.

Restricting the number of read and write instructions to 32 each helps maintain the encoding of the header chunk within 128 bytes. Fewer than 32 would increase the pressure on the register allocator in the compiler and greater than 32 would necessitate a larger header chunk that not only increases the code footprint in memory, but also the I-cache capacity requirements. Restricting the number of loads and stores in the block to 32 reduces the number of entries needed in the load-store queue and reduces its complexity. The explicit encoding of read and write instructions in the block header helps the resolution of inter-block data dependences. Using this encoding a new block can quickly identify if any of the previous blocks produce its input registers at fetch time; otherwise it must wait until all previous blocks have completed their execution, reducing the overall performance.

4.2 TRIPS Prototype Microarchitecture

The TRIPS prototype chip consists of two processors adjacent to an array of non-uniform cache access (NUCA [80]) L2 cache banks. Figure 4.3 provides an organizational overview of the TRIPS chip. Each of the two processors implements the TRIPS microarchitecture described in Chapter 3. Each processor issues 16 out-of-order operations per cycle, buffers 1024 in-flight instructions, and contains 80 KB

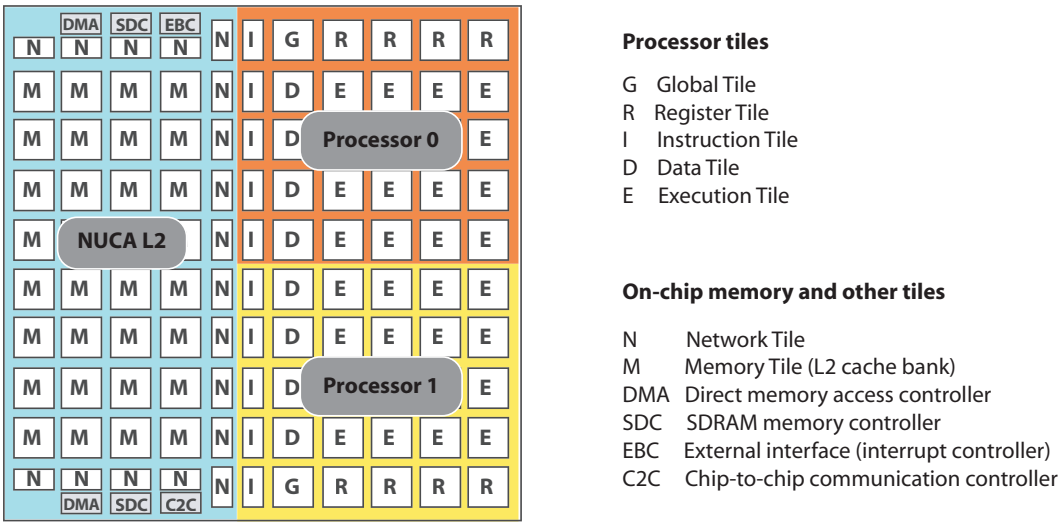


Figure 4.3: TRIPS chip overview.

of L1 instruction cache and 32 KB of L1 data cache. A processor can utilize all these resources for one thread or up to four simultaneously multi-threaded (SMT) threads, thus providing the ability to exploit both instruction-level parallelism and thread-level parallelism. The two processors share the L2 memory system. The sixteen L2 cache banks together provide a 1 MB L2 cache in the prototype chip.

As described in Chapter 3, the chip implementation embraces a few key design principles for managing complexity and enhancing design productivity: a) use of a small number of heterogeneous components, b) component re-use, and c) avoidance of global wires. Accordingly, the processor microarchitecture consists of multiple heterogeneous tiles—execution units, register file banks, data and instruction cache banks—and connects them using a set of simple data and control networks. Likewise, the on-chip memory system is composed from multiple memory tiles residing on a switched-network. Each tile has a small area, ranging from 1–9 mm^2, so that local wires—in low-level metal layers—can accomplish the communication needs within the tile. For inter-tile communication, global wires are

replaced by point-to-point networks. Such an organization enhances scalability to larger implementations that support more tiles and larger network topologies, without significant re-design of each tile.

4.2.1 Processor Tiles and Networks

Each processor core is implemented using 30 tiles that belong to five unique types: one global control tile (GT), 16 execution tiles (ET), four data tiles (DT), four register tiles (RT), and five instruction tiles (IT). Table 4.2 presents the composition of each tile. Each ET consists of an integer and floating point unit, a 64-entry reservation station, and a standard single-issue execution pipeline. Each RT contains a portion of the architecture and physical register file. Each DT consists of a data cache bank, cache miss handling logic, load/store queues, and a 1-bit dependence predictor to predict the dependences among in-flight memory loads and stores. The ITs comprise the primary memory system for instructions. The GT sequences the overall execution of a program. The particular configuration shown in the table provides a 16-wide issue, 1024-instruction window TRIPS processor.

As described in Chapter 3, the reservation stations in the execution array are partitioned into equal portions in each ET. These portions are aggregated across all the ETs to form a frame, on which a single block can be mapped and executed. The prototype microarchitecture supports eight frames, each containing eight reservation station entries per ET, thus providing a window of 128 slots across the entire array of ETs. The hardware maps each new block in an available frame and executes it. The hardware can be configured to run in either single-threaded mode or simultaneous multi-threaded mode. In the single-threaded mode of operation, up to eight blocks belonging to the same thread can be in flight simultaneously, seven of them speculatively. In the multi-threaded mode of operation, each thread can have up to two blocks in flight, one of which is speculative. Control registers within the

Tile	Composition
GT	Block management state supporting eight blocks in single-threaded mode and two blocks for each of 4 SMT threads in multi-threaded mode, processor status and control registers, interface to on-board master control processor, L1 I-cache tags, 128 TRIPS blocks, 2-way set-associative, 16-entry, fully-associative instruction TLB, 84 Kbit next-block predictor that includes a local/global adaptive tournament exit predictor with speculative updates, branch/call target buffers, branch type and return predictors, 3-cycle predict and update operations, 2-cycle repair operation.
RT	Four 32-register banks, one each for 4 SMT threads, 64 physical registers, eight for each in-flight block, inter-block register dependence check logic.
IT	16 KB bank, 64-byte lines, one cycle hit latency, 128-bit input and output interfaces to the OCN, transfers 64 bytes every five cycles with L2 cache in each direction.
DT	2-way, one-cycle 8 KB L1 cache bank, with 64-byte cache lines, cache-line interleaving among 4 DTs, one 256-entry LSQ, one-entry coalescing write buffer, 16-entry, fully-associative data TLB, MSHRs supporting up to 16 requests to up to four cache lines, memory-side, 1024-entry, single-bit dependence predictor to predict the dependences between program stores and loads.
ET	64-entry reservation station holding decoded instructions, each with two 64-bit data operands, and one-bit predicate operand, a five-stage, single-issue execution pipeline, one integer unit and one FP unit, single cycle basic integer operations, 3-cycle, pipelined integer multiply, 24-cycle, non-pipelined integer divide, 4-cycle FP operations, no support for FP divide and sqrt
Processor	30 tiles: 1 GT, 4 RTs, 5 ITs, 4 DTs, 16 ETs 16-wide issue, 1024-entry window

Table 4.2: Composition of the processor tiles.

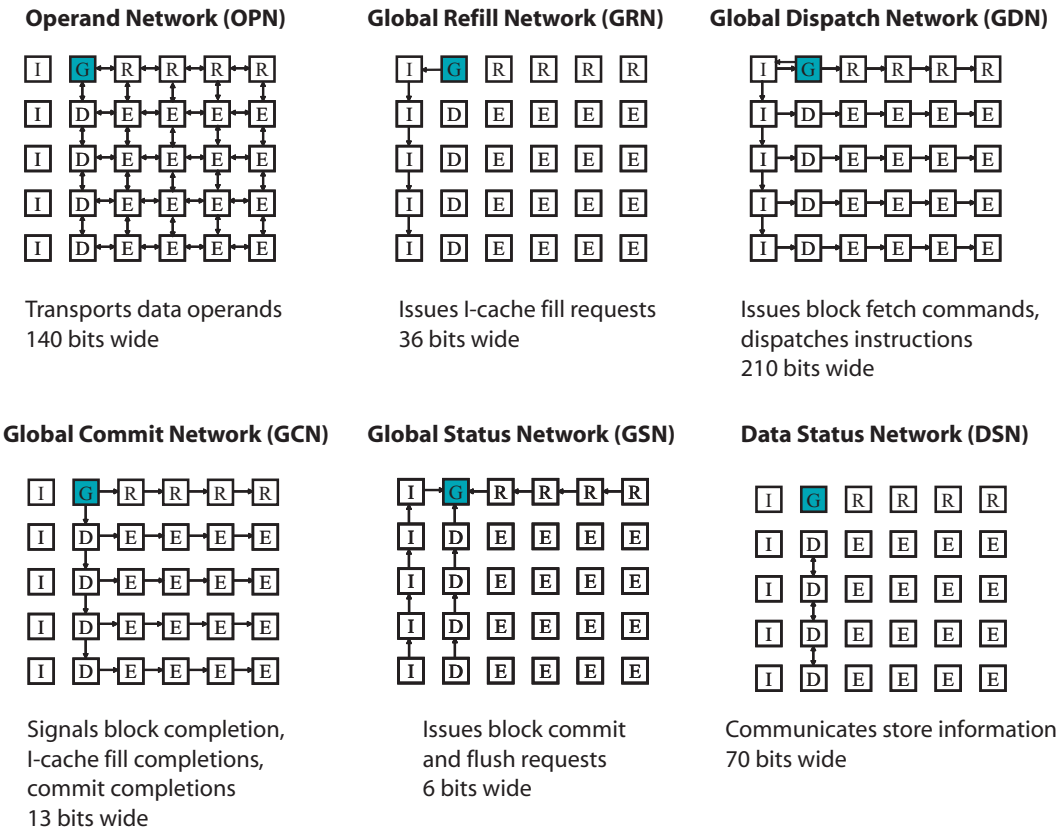


Figure 4.4: TRIPS microarchitectural networks.

GT configure the processor into one of these two modes of operation.

Seven different point-to-point networks connect the tiles together. Each link on a network connects only nearest neighbors—north, east, south, or west in a Manhattan topology—and incurs a one-cycle transmission latency. Figure 4.4 shows the connections in six of these networks. It also provides a brief summary and the width of each network. One network, called the global dispatch network (GDN), is used for dispatching instructions from the ITs to the ETs and the RTs. Another network, called the global refill network (GRN), is used for directing the ITs to fill missing instruction cache lines. Three other networks, global control network (GCN), global

status network (GSN), and external store network (not shown in Figure 4.4) are used for exchanging global control information back and forth between the GT and other tiles. Finally, the data status network (DSN) connects the DTs together to communicate store information among them. There is no flow control in any of these networks, and the consumer is expected to sample a link each cycle and source the data immediately.

The major network in the processor, however, is the operand network (OPN). It connects all tiles except the ITs in a 5 × 5, worm-hole routed, mesh network and communicates data operands between them [66]. Each link consists of separate control and data channels and can transfer one 64-bit data operand each cycle. Every packet sent on the network consists of two flits—control and data. The control flit leads the data flit by exactly one cycle and prepares the consumer to receive and use the data operand in the following cycle. This mechanism reduces the latency of execution of dependent instructions mapped on different tiles. The OPN avoids deadlocks using guaranteed reception of a delivered packet and a dimension-ordered routing policy—each packet first traverses vertically to the destination's row and then traverses horizontally to the destination. The OPN uses on-off flow control to implement loss-less FIFO-ordered delivery of packets between any pair of tiles on the network.

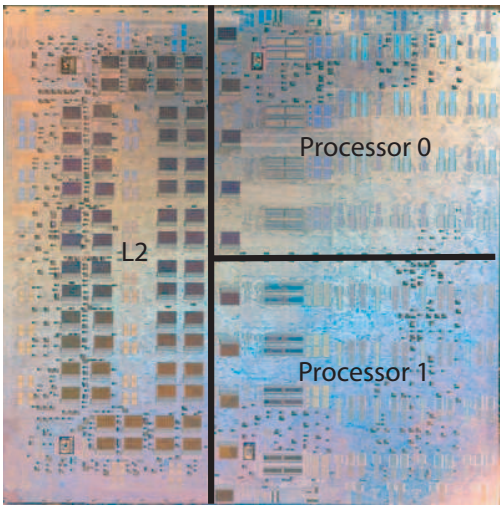
4.2.2 Secondary Memory System

The TRIPS prototype chip contains a 4-way, 1 MB, on-chip L2-cache, implemented using 16 memory tiles (MTs) as shown in Figure 4.3. Each MT contains a 64 KB data bank, which may be configured as a cache bank or as a byte-addressable scratch-pad memory. The network tiles (NTs) surrounding the MTs translate memory addresses to determine where the data for a particular address may be found. The NTs and MTs are clients on another network called the On-Chip Network (OCN), which is

a 4×10, two-dimensional, worm-hole routed network [65]. The OCN also interfaces with each of the ITs on the edge of the TRIPS processors to provide high-bandwidth L2 cache access. Each IT/DT pair on the same row share the OCN port to inject cache fill requests to the L2 cache. A processor may request up to five cache line fills each cycle through its five OCN output ports. The NTs translate the address for a fill request and the OCN routers eventually transmit the request to the MT that can service the request.

4.2.3 On-Chip Controllers

The TRIPS chip also includes several controllers that are attached to the OCN for implementing different system-level functionalities. The two SDRAM controllers (SDC) each connect to a separate 1GB SDRAM module. The chip-to-chip controller (C2C) connects a TRIPS chip to other chips in the system. The two direct memory access (DMA) controllers are used to transfer data directly to and from any two portions of the physical address space, including the SDRAM and any memory-mapped, on-chip storage such as the L2 cache and the processor registers. Finally, the external bus controller (EBC) provides an interface to an on-board PowerPC processor, to which the TRIPS processors act as slave co-processors. To simplify the design, the prototype chip off-loads all operating system functionality to the PowerPC processor and a host PC connected to the TRIPS motherboard. A host PC running a commodity OS downloads a program to run on to the address space visible to the TRIPS chip. The chip runs the program until it encounters a system call, upon which it relinquishes control to the host PC to service the system call. After the host PC services the system call, it resumes the execution of the TRIPS chip.



Fabrication process	130 nm ASIC
Die size	18.3 mm x 18.37 mm (336 sq. mm)
Transistor count	170 million
Package	47 mm x 47 mm ball grid array
Pin count	626 signals, 352 Vdd, 348 Gnd
Routed cells	6.1 million
Routed nets	6.5 million
Total wire length	1.06 Km
Worst case clock period	4.5 ns

Figure 4.5: TRIPS die photo. Picture shows the boundaries of the two TRIPS processors and the secondary cache.

4.2.4 TRIPS Chip Implementation

The TRIPS chip was implemented in the IBM Cu-11, 130 nm ASIC process. It consists of more than 170 million transistors in a chip area of 18.30 mm by 18.37 mm. Figure 4.5 shows the die photograph of the chip and the boundaries of the processors and the L2 cache superimposed over the photograph. It also shows various physical attributes of the chip implementation. The TRIPS chip taped out in August, 2006 and first silicon was delivered in October, 2006. After initial electrical testing, the first TRIPS chip was mounted onto the prototype system boards and brought to life in early November, 2006. Figure 4.6 shows photographs of the delivered TRIPS chip part and the TRIPS motherboard. The TRIPS chip is mounted on a separate daughtercard, which offers plug-and-play testing of individual TRIPS chips. The daughtercard is mounted on to the motherboard which interfaces with a host PC.

At the time of writing this dissertation, in Spring 2007, all the major functionalities of the TRIPS prototype system have been tested successfully. The chips

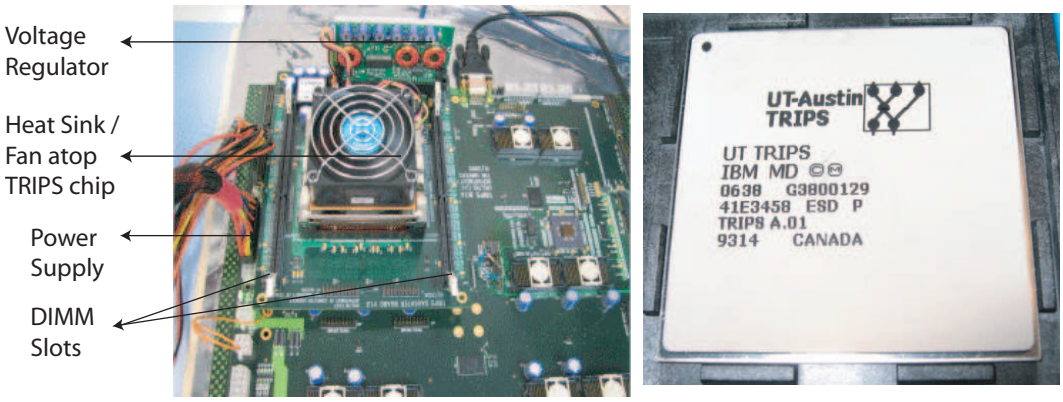


Figure 4.6: Picture of TRIPS motherboard and package.

are able to run several workloads, including single-threaded workloads such as SPEC CPU2000, successfully. In addition, they are able to run SMT threads and multi-threaded MPI workloads successfully. The peak clock rate of the functional chips was observed to be 366 MHz, which is in tune with the taped-out worst-case clock period of 4.5 ns and improvements from nominal process corners of the fabrication technology. Current hardware measurements indicate a peak power consumption of 45 W at a clock frequency of 366 MHz for the entire TRIPS chip. Most of this power is spent in the clock distribution network and idle dynamic power.

Full-custom implementations of the TRIPS chip will offer higher clock rates than the ASIC prototype implementation. Chinnery et al. discuss a number of limitations of ASIC design flows, which when addressed in custom design can offer dramatic improvements in clock rates [33]. The modular organization of the TRIPS microarchitecture and short point-to-point interconnections greatly facilitate a high frequency design as they mitigate the effect of wire delays. However, logic paths within various tiles and those that span neighboring tiles will undoubtedly need to be re-designed to support high clock rates. Nevertheless, we expect the TRIPS design to support competitive clock rates in a custom design.

4.3 Development Effort

Developing a working silicon prototype, especially for distributed microarchitecture with an unproven design and ambitious performance goals, is an enormous undertaking. Many people have contributed to this effort over the course of three years. In this section, I describe the design and implementation effort for the TRIPS prototype chip and highlight my specific roles and contributions.

4.3.1 Overall Effort

The design and implementation of the TRIPS prototype chip was accomplished with a team of 12 students, two staff members, and two faculty members at the University of Texas at Austin (UT) and an ASIC design team at IBM, Austin, TX. The UT team was responsible for a post-synthesized netlist for the chip, whereas the IBM team provided the design libraries and was responsible for most of the physical design and the tapeout process. The high-level architectural exploration started in 2000 at UT with two students and ramped up to nine students in the spring of 2003. The entire design and implementation effort for the prototype chip was led by one professional engineer at UT from the spring of 2003, when the detailed chip specification started, until tapeout in the summer of 2006. The specification was progressively refined using detailed performance simulators until early 2004, when RTL design and entry began. The team ultimately peaked in summer of 2004 with the addition of three more students. The RTL design, implementation, and verification was completed over a period of 18 months starting from 2004 until the end of 2005. One-fourth of the effort at UT was spent in RTL design and implementation, two-thirds were spent in verification, and the remainder in physical design. The whole chip was implemented with 11 different modules.

The tiled organization of the TRIPS processor enabled a hierarchical verification strategy. Each module design team was responsible for the complete verification

of the logic within the module. The verification methodology involved self-checking randomized tests to cover as many events as possible. Since the tiles are relatively small in relation to the entire processor, this methodology yielded fast simulation times and helped uncover a majority of the bugs. The modules were then instantiated in the higher level components, namely the processor and the L2 system, and the components were verified for functional and performance correctness using random test vectors. At this level, the verification focused on the correct execution of complete programs and the protocol implementation in different tiles. Finally, the top-level chip module that instantiates the processor, L2, and various on-chip controllers was verified for functional correctness. At this level, where the simulation is quite slow, the verification effort largely focused on diagnostic tests and correct access of all on-chip state from the host PC. Of the total verification effort, roughly a half was spent in the module-level verification, and the remainder was spent equally between the component-level and the chip-level verification effort.

The TRIPS board and daugthercard were jointly designed and tested by UT and University of Southern California/Information Sciences Institute (USC/ISI East). This effort was completed over a period of 12 months. After delivery of the first TRIPS chips, a team of four students and one staff member at UT verified the correct functionality of the delivered parts over a period of three months.

Concurrently with the hardware design and implementation, a team of five students, three staff members, and two faculty members began the development of the TRIPS software toolchain. At the time of writing this dissertation, the toolchain is able to compile all SPEC workloads correctly, and is being ramped up for generating high quality code.

4.3.2 My Contributions

The high-level architecture, microarchitecture, and the execution model for the TRIPS architecture were jointly developed by the author and Karthikeyan Sankaralingam. I specified the TRIPS ISA in collaboration with Robert McDonald, Karthikeyan Sankaralingam, Doug Burger, and Steve Keckler. I developed the ILP techniques in the microarchitecture, including register renaming and the block control protocols that provide various services for performing execution. Karthikeyan Sankaralingam and I jointly led the implementation of *tsim-proc*, which is the high-level, detailed performance model for the TRIPS prototype processor.

I designed, implemented, and verified the GT along with Nitya Ranganathan, who implemented the next-block predictor. I also collaborated with her in verifying the correct functionality of the predictor. I led the performance verification of the prototype processor, and along with Nitya Ranganathan, correlated the performance of the RTL implementation with *tsim-proc* to within 4%. During the functional verification of the top-level chip module, I developed several targeted tests for verifying the correct functionality of the processor. During the hardware bringup efforts, I was instrumental in tracking system software bugs, whose resolution enabled the correct and complete execution of simple workloads.

I also developed the benchmark simulation infrastructure for evaluating the performance of the TRIPS prototype processor. I hand-optimized several benchmarks in the TRIPS intermediate assembly language to identify opportunities for compiler optimizations and form an evaluation suite of fully optimized benchmarks. Finally, I developed detailed performance analysis tools, including *tsim-critical*, which identifies the performance bottlenecks in the prototype ISA and the microarchitecture.

Discussion: In this section, we described the overall implementation of the TRIPS prototype chip. In the remainder of this chapter, we describe two aspects of the implementation in depth—block control and performance verification. Since the prototype processor microarchitecture is our vehicle for performance evaluation, an understanding of the block control mechanisms is crucial to understanding the overall performance of the TRIPS architecture. Section 4.4 describes the implementation of the block control mechanisms in the TRIPS prototype processor. It first describes the logic blocks in the GT, then describes the distributed protocols for various block operations, including fetch, flush, and commit. Section 4.5 describes the performance verification of the prototype processor and the key microarchitecture events whose latency and throughput affect the overall performance.

4.4 Block Control

The distributed execution of a single block involves all the tiles in the microarchitecture. To execute a new block, the GT must first allocate a free frame. The ITs then dispatch instructions to the ETs, where the instructions execute in a dataflow fashion. Operand values trickle through the microarchitecture from tile to tile and eventually the block outputs reach the RTs and the DTs. The GT must then detect completion, commit the outputs, and deallocate the resources utilized by the block. Such distributed execution requires solutions to two major challenges: a) controlling the individual operations of the distributed tiles, and b) managing the execution state of all in-flight blocks. In the TRIPS processor, the GT accomplishes both of these functions.

The GT implements all block operations by maintaining control state on behalf of the entire processor and using set of master-slave distributed control protocols, including fetch, flush, and commit, running over the control and data networks. For design simplicity and high performance, these protocols must satisfy

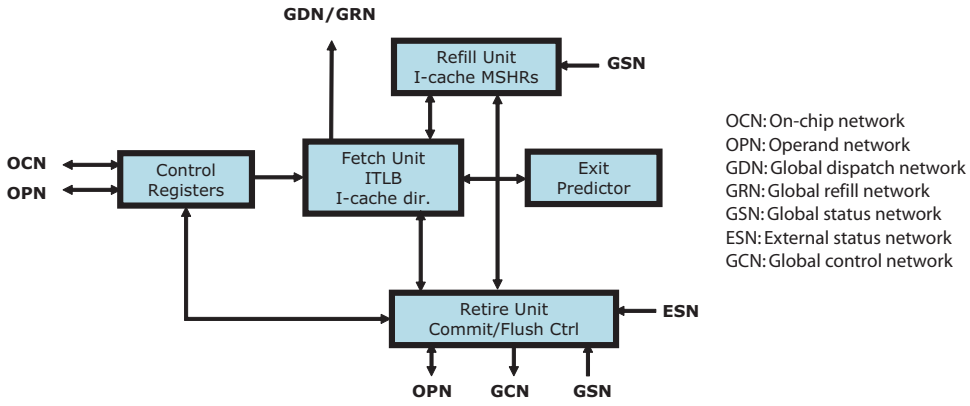


Figure 4.7: High-level organization of the GT.

two key properties. First, any control state maintenance must attain a balance between centralization—for minimizing replication—and distribution—for maximizing concurrency. Second, since transporting signals across different tiles involves high latency, the protocols must be latency tolerant and must overlap with each other as much possible.

4.4.1 GT Implementation

The GT implements all of its logic functions using four major sub-units: the fetch unit, refill unit, retire unit, and the exit predictor. Figure 4.7 shows the high-level organization of these sub-units. In the subsequent paragraphs, we describe each of them in detail and compare them with their counterparts in conventional processors.

Fetch Unit

The fetch unit consists of a TLB (Translation-Lookaside Buffer) and a directory of the blocks that are resident in the I-cache. In addition, it contains the program counters (PC) for each thread and control registers that are used to configure the execution of each block.

Tags for cached blocks	GT
Header chunk	IT0
Instruction chunk 0	IT1
Instruction chunk 1	IT2
Instruction chunk 2	IT3
Instruction chunk 3	IT4

Table 4.3: I-cache storage of a block in the TRIPS processor.

Name	Description	Bit width
V	Valid block	1
L	LRU information	1
PTAG	Physical tag of the block's address	27
H	Meta information for the block	40

Table 4.4: An entry in the I-cache directory.

I-cache Directory: Table 4.3 provides an overview of how the ITs stripe the instructions of a single block. Each IT caches one chunk, which corresponds to the instructions mapped to the same row of ETs or RTs. The I-cache directory contains a listing of all blocks that are currently resident in the I-cache. Table 4.4 depicts the components of each entry in the directory. The directory consists of 128 entries, and is organized in a 2-way set-associative fashion. Each entry identifies a unique cached block and also stores a portion of the meta information associated with the block. The directory is virtually indexed, and entries are evicted and replaced in a LRU fashion. This configuration supports a 2-way, set-associative caching of 128 TRIPS blocks. A larger size could not be supported due to area restrictions in the ITs.

The I-cache directory is similar to the tag array in conventional caches. In the TRIPS processor, the GT maintains a single array on behalf of all the ITs. An alternate design could maintain the tag arrays within each IT. This technique avoids centralized control for the I-cache operations. However, it requires special hardware to keep the tag arrays consistent, as a single block is striped across all ITs, and each IT operates independently of the others. A centralized directory provides a consistent view of the cached blocks and avoids scenarios where portions of a block are resident in one IT, but not in others. To evict a block from the cache, the GT simply invalidates the corresponding entry in the I-cache directory without notifying the ITs. The tag array in each IT can be eliminated, thus simplifying the

implementation in both the GT and ITs.

Instruction TLB: A set of sixteen registers provide the translations of virtual addresses of blocks to physical addresses. Similar to the I-cache directory, implementing the TLB registers inside the GT avoids redundant implementation in the ITs. Each register defines the size and read/execute access attributes of a memory page. The minimum size of a memory page is 64 KB and the maximum size is 1 TB. Instruction memory pages may be marked as uncacheable in the L1. A block in such a page will never be filled into the I-cache. A miss in the TLB or an access protection violation will result in an exception being generated. The TRIPS system software manages the TLB and must be designed around the expectation that TLB misses are non-existent or infrequent. Due to the support for large physical pages, we expect 16 TLB entries to be adequate for a majority of the applications on the TRIPS prototype system.

Refill Unit

The refill unit maintains the status of pending I-cache fill operations. The TRIPS processor supports outstanding fills for up to four blocks, but at most one per thread. Table 4.5 shows the state that the GT tracks for each pending fill. The state includes information such as the I-cache set and the way being filled, whether the fill has completed or not, and the meta header information for the block being filled. This pending state is similar to the I-cache MSHR (Miss Status Handling Register) state in conventional processors. However, in the TRIPS processor it serves the purpose of managing the distributed fill operations in the ITs.

Retire Unit

The retire unit consists of the retirement table which tracks the execution state of all blocks in flight. It is also responsible for initiating the flush, commit, and deal-

Name	Description	Bit width
V	Valid refill	1
S	Set in the cache being filled	6
W	Way in the set being filled	1
TID	Thread corresponding to the fill	2
PTAG	Physical tag of the block's address	27
F	Fill already flushed/cancelled	1
C	Filled completed	1
Ca	Block L1 cacheable or not	1
H	Meta information for the filled block	40

Table 4.5: State tracked for each pending fill.

Name	Description	Bit width
V	Valid block	1
O	Oldest block in thread	1
Y	Youngest block in thread	1
BADDR	Virtual address of the block	40
PADDR	Predicted address of the next block	40
RADDR	Actual resolved address of the next block	40
RC	Registers completed	1
SC	Stores completed	1
BC	Branch completed	1
RCOMM	Registers committed	1
SCOMM	Stores committed	1
E	Exception in block	1

Table 4.6: State tracked for each block in the retirement table.

location of the blocks in flight. Table 4.6 shows the details of the state maintained for each block. Most of this state is updated locally by the GT when it starts various block-level operations. The rest of the state is updated when the GT receives notifications on the control networks from other tiles. Each cycle the GT monitors the state for every block and initiates the flush, commit, or deallocation operations as necessary. For example, it initiates a flush of a valid (`V`) block if the resolved address (`RADDR`) for its next block does not match the predicted address (`PADDR`). However, if a valid block is the oldest in a thread (`O`), all of its outputs have been received (`RC, SC, BC`), and there are no control flow mispredictions or exceptions (`E`), the GT initiates a commit for the block.

The retirement table is similar to the reorder buffer (ROB) in conventional processors. However, this table does not track the status of individual instructions. It has only one entry for each block, thus containing far fewer entries than a conventional ROB.

Next-Block Predictor

The next-block predictor predicts the address of the next block to execute from a single thread. It uses both local and global history information and employs a tournament-style prediction similar to the Alpha 21264 predictor. The predictor state amounts to a total of 84 Kbits and sustains competitive accuracies compared to the Alpha 21264 predictor [126]. The predictor performs three major operations—*predict, update, and repair*. Predict provides a prediction for the next block. Update modifies the predictor tables with the information from a committing block. Repair corrects any predictor state modified by incorrect speculation. The predict and update operations each consume three processor cycles, while the repair consumes two cycles. None of the operations are overlapped with any other.

Physical Implementation

The GT occupies roughly 2% of the area in each processor. It is implemented in a 3.4 mm $\times$ 0.9 mm rectangular tile. Table 4.7 provides a breakdown of the area consumed by different units within the GT. The column labeled *Cell Count* shows the number of placeable instances in each unit, which provides a relative estimate of the complexity in each unit. The column labeled *Array Bits* indicates the number of bits in the dense register and SRAM arrays in each unit. The final column shows the fraction of the GT area occupied by each unit. As shown in the table, the next-block predictor consumes nearly 50% of the area in the GT. The fetch and retire sub-units consume 19% and 11% of the area respectively. The OPN router inside

Unit	Cell Count (× 1000)	Array bits (× 1000)	% Area
Fetch	10.0	8.6	19.1
Retire	12.1	0.0	11.3
Next-block predictor	9.7	84.1	48.2
OPN router	14.1	0.0	13.7
Other	7.2	0.0	7.7
Total	53.1	92.7	100.0

Table 4.7: Area breakdown for the GT.

the GT occupies 14% of the GT area. The refill unit and other miscellaneous logic consume the rest of the area.

4.4.2 Block Operations

The GT uses the four logic blocks described in the previous section to implement four block-level operations: a) I-cache fills, b) block fetch and dispatch, c) block commit, and d) block flush. The subsequent paragraphs describe each of them in detail.

I-cache Fills

The I-cache fill of a block happens in two steps – *fill* and *update*. In the fill step, the instruction bits are fetched from the secondary cache and buffered in an auxiliary structure in the IT called the fill buffer. In the update step, the instruction bits are read from the fill buffer and written into the I-cache banks. The refill protocol performs the fill operation. The fetch protocol described in the next section performs the update operation.

Figure 4.8 depicts the different events during the execution of the refill protocol. It begins with the GT sending the physical address of the block on the GRN interface (cycle 5). During the preceding cycles (0–2), the GT computes the address of the block to refill. The GT performs a TLB translation, looks up the I-cache

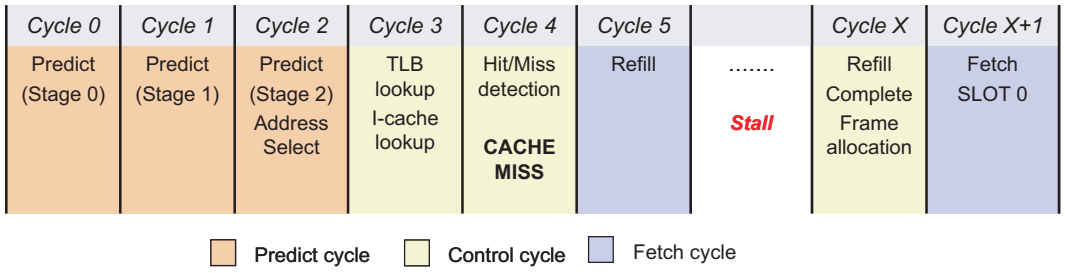


Figure 4.8: Refill pipeline.

directory, and detects a miss in cycles 3 and 4. It begins the refill operation in cycle 5. Each IT subsequently receives the refill command and independently launches several transactions with the secondary memory using the OCN to fetch its chunk of the block. When an IT completes its fill operation, it sends a notification signal upwards to its top neighbor using the GSN. The IT sends such a notification signal only if it has already received a similar signal from its bottom neighbor. The top-most IT notifies the GT. Thus the refill completion signal daisy-chains all the way from the bottom-most IT to the GT, and the GT eventually receives one notification that marks the completion of the entire refill operation.

The centralized I-cache tags and the refill protocol allow the GT to control the operation of the ITs effectively. Occasionally, the GT may chose to discard a block saved in the fill buffers without updating the I-cache. It does so by simply not initiating an update step. An alternate design for the refill protocol might merge the fill and update steps and eliminate the need for fill buffers. However, we observed that branch mispredictions often produced addresses that did not correspond to any legal block. The resulting spurious refills pollute the I-cache and evict other blocks that are currently in the working set of the program. This problem is particularly severe in the presence of small blocks, which fill the I-cache with NOPs and reduce the program working set that is resident in the cache. Occasionally, branch mispredictions also resulted in correct prefetching refills. However, we observed that the

pollution resulting from spurious refills outweigh the benefits of serendipitous refills. This observation motivated the separation of the refill protocol into the distinct fill and update steps.

Fixed-size header and instruction chunks help simplify the implementation of the block refill protocol. They result in fixed offsets for the individual chunks and a simple deterministic partitioning of a block's instructions among the different ITs. Variable-sized chunks would increase the complexity of the block refill protocol considerably, as the address offset for the chunk cached by each IT is unknown prior to the fill. They necessitate either a centralized fill operation, or redundant fill operations at each IT that must later be synchronized. Fixed size chunks also simplify the block fetch protocol as it results in fixed latency to fetch every block and reduction in the I-cache tag state maintained at the GT.

Block Fetch

The GT initiates a fetch protocol to distribute the instructions from the IT banks to the execution units. Figure 4.9 shows the different events during the fetch protocol. Similar to the refill operation, the GT performs a TLB translation, looks up the I-cache directory, and detects a cache hit in cycles 3 and 4. It allocates a free frame for the block in cycle 4 and begins the fetch protocol in cycle 5 by issuing a command on the GDN. The command includes the cache index to fetch from, the address of the block, and the frame identifier allocated for the block. In addition, the GT also instructs the ITs to perform the update step of a refill operation, if the fetch resulted from a preceding refill. Since the instruction distribution for the block from a single IT is pipelined over a total of eight cycles, the GT sends seven more pipelined indices, in cycles 6–12, to initiate the block fetch.

Figure 4.10 shows the timing of block instruction distribution to all of the ETs. The GDN has a 128-bit (four-instruction) wide channel that is routed from

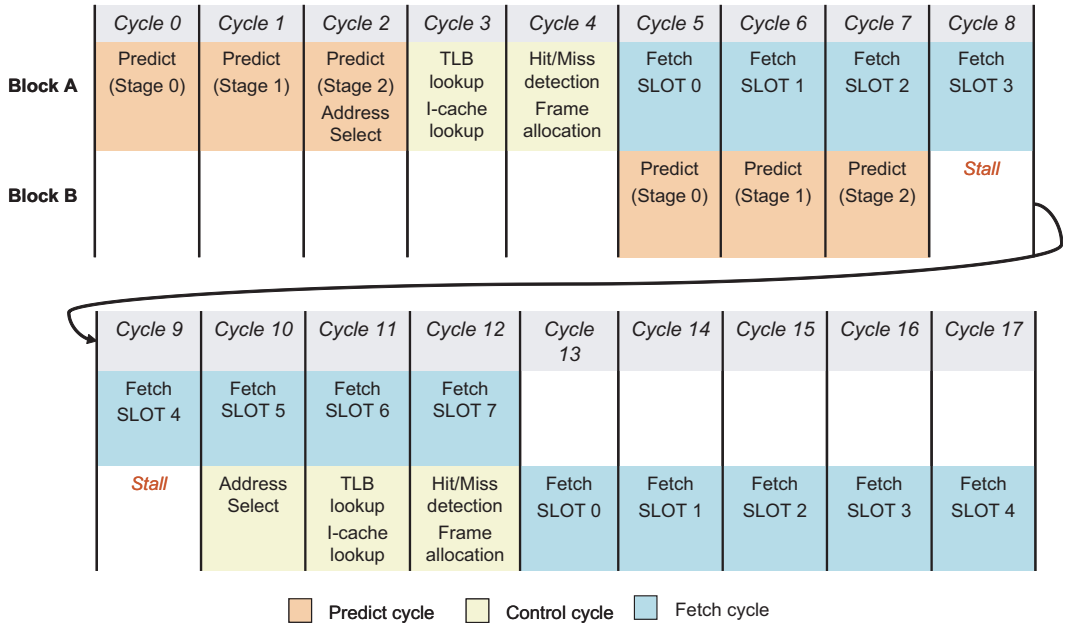


Figure 4.9: Fetch pipeline.

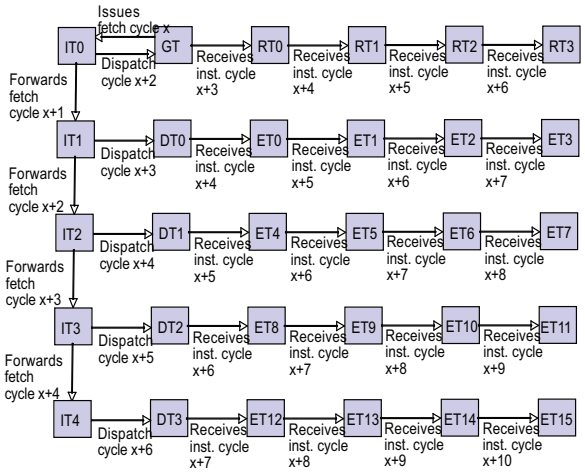


Figure 4.10: Timing of block fetch and instruction distribution. The figure depicts the delivery time of the first instruction at each ET/RT. Each tile continues to receive a new instruction each cycle for the next seven cycles.

the ITs, through the DTs, and to the ETs. Each cycle of dispatch, the ITs send out four instructions on the GDN, one for each ET in its row. Assuming that the block dispatch command is issued by the GT in cycle X, the closest ET (upper-left corner) receives its first instruction for that block in cycle X+4, and continues receiving one instruction per cycle until it receives its last instruction for the block in cycle X+11. The farthest ET (`ET15`) receives its first instruction for the block in cycle X+10, and its last in X+17. The distribution of the `read` and `write` instructions to the RTs in the top row proceeds in a similar fashion.

While the latency to complete a distributed fetch operation is relatively large (18 cycles), multiple block fetches can be pipelined, so that at steady-state peak operation, each ET receives one fetched instruction per cycle with no fetch bubbles in between blocks. Figure 4.9 shows how the fetches for two blocks are pipelined. The GT consumes eight cycles to initiate the fetch of the first block, starting from cycle 5. In parallel, a prediction is made and the fetch of the next block is set up during the cycles 5–12. The fetch of the second block starts at cycle 13 and lasts until cycle 20. Running at peak, the machine can issue fetch commands every cycle with no bubbles, beginning a new block fetch every eight cycles.

The distributed fetch protocol in the TRIPS processor provides significantly higher fetch bandwidth compared to conventional processors. Each IT independently fetches and distributes instructions for its row, which provides a peak fetch rate of 16 instructions each cycle—4 rows $\times$ 4 instructions per row per cycle—matching the peak execution rate of the processor. Managing the free list of frames in the GT and propagating the allocated identifier along with every fetch reduces the complexity of frame management in other tiles.

One downside of the implementation is the fact that it tightly couples the predictor operations and the fetch protocol operations in one single pipeline. In steady state, the three cycles for predict and three cycles for update can fully overlap

with the 8 cycles of fetch required for one block. Thus there are no bubbles in the fetch pipeline, enabling a new block fetch every eight cycles. Occasionally, predictor operations may cause bubbles in the fetch pipeline. For example, in Figure 4.9, a 3-cycle update operation starting in cycle 4 and a 2-cycle repair operation starting in cycle 7 will delay the predict operation for the second block until cycle 9. The fetch of block B will therefore not start until cycle 14, introducing a bubble in the pipeline.

An alternate design could have completely decoupled the prediction pipeline from the fetch pipeline using a *Fetch Target Buffer* [128]. That design offers two advantages. First, multiple refills can be initiated well ahead of a fetch, thus prefetching several blocks into the I-cache. Second, stalls in the predict pipeline are less likely to affect the fetch pipeline. It, however, incurs additional hardware complexity, which did not appear to be worth the benefits during the implementation.

Block Flush

Because TRIPS executes blocks speculatively, a branch misprediction, a load/store ordering violation, an exception, or an external interrupt causes pipeline flushes. These flushes are implemented using a distributed protocol. The GT is first notified when a mis-speculation occurs, either by detecting a branch misprediction itself or via a GSN message which indicate exceptions and memory-ordering violations. The GT then initiates a flush wave on the GCN that propagates to all of the ETs, DTs, and RTs, taking one cycle per hop through the array. The flush wave includes a block identifier mask indicating which block or blocks must be flushed. The processor must support multi-block flushing because all speculative blocks after the one that caused the mis-speculation must also be flushed.

The GT invalidates all the state corresponding to the flushed blocks in the retirement table and stops pending fetches for flushed blocks. Other tiles also inval-

idate the state corresponding to the flushed blocks. The DTs mark pending cache miss operations from the flushed blocks so that they can be discarded when they complete eventually. All tiles must hold the flush command received from the GT for two additional cycles to invalidate state for the flushed blocks cleanly. This condition is required because the GCN has a shorter path to the DTs, RTs, and ETs from the GT than the GDN, which allows a fetch wave for a flushed block to arrive at a tile up to two cycles after the flush wave. The flush window in the DTs, RTs, and the GTs therefore lasts three cycles (including the cycle it receives the flush) to cancel any trailing fetch wave. The GT may issue a fetch command for a new block three cycles after the flush. The intervening cycles are consumed by the fetch pipeline to access the I-cache directory and TLBs before initiating the fetch.

Block Commit

Block commit is the most complex of the distributed control protocols in the TRIPS processor, since it involves the three phases illustrated in Figure 4.11: block completion, block commit, and commit acknowledgment. In phase one, a block is complete when it has produced all of its outputs, the number of which is determined at compile-time and consists of up to 32 register writes, up to 32 stores, and exactly one branch. After the RTs and DTs receive all of the register writes or stores for a given block, they inform the GT using the Global Status Network (GSN).

Each RT counts the number of writes it receives and detects when it has received all of the expected writes. The expected number of writes at an RT is known at compile time, as the writes are encoded explicitly in the write instructions. When an RT detects that all writes have arrived, it informs its west neighbor. The RT completion message is daisy-chained westward across the RTs, until it reaches the GT indicating that all of the register writes for that block have been received.

A branch instruction sends its result to the GT using the OPN. The GT

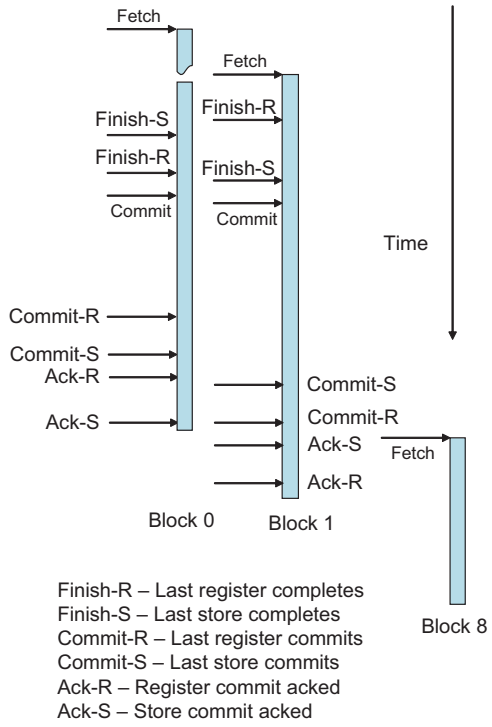


Figure 4.11: Timing of block commit protocol.

detects branch completion when it receives the branch notification.

Detecting store completion is more difficult since each DT cannot know *a priori* how many stores will be sent to it. To enable the DTs to detect store completion, we implemented a DT-specific network called the Data Status Network (DSN). Each block header contains a 32-bit *store mask*, which indicates the memory operations in the block that are stores. This store mask is sent to all DTs upon block dispatch. When an executed store arrives at a DT, its 5-bit LSID and frame identifier are sent to the other DTs on the DSN. Each DT then marks that store as received, even though it does not know the store's address or data. Thus, a load at a DT learns when all previous stores have been received across all of the DTs. The nearest DT notifies the GT when all of the expected stores of a block have ar-

rived. When the GT receives the GSN signal from the closest RT and DT, and has received one branch for the block from the OPN, the block is complete. Speculative execution may still be occurring within the block, down paths that will eventually be nullified by predicates, but such execution will not affect any block outputs.

During the second phase (block commit), the GT broadcasts a commit command on the GCN and updates the block predictor. The commit command informs all RTs and DTs that they should commit their register writes and stores to architectural state. To prevent this distributed commit from becoming a bottleneck, we designed the logic to support pipelined commit commands. The GT can legally send a commit command on the GCN for a block when a commit command has been sent for all older in-flight blocks, even if the commit commands for the older blocks are still in flight. The pipelined commits are safe because each tile is guaranteed to receive and process them in order. The commit command on the GCN also flushes any speculative in-flight state in the ETs and DTs for that block.

The third phase acknowledges the completion of commit. When an RT or DT has finished committing its architectural state for a given block it signals commit acknowledge on the GSN. A tile sends a commit acknowledge only after it commits all architecture state it is responsible for *and* after it has received an acknowledge signal from its neighbor on the GSN. When the GT has received commit completion signals from both the RTs and DTs, it knows that the block is safe to deallocate, because all of the block's outputs have been written to architectural state. When the oldest block has acknowledged commit, the GT initiates a block fetch and dispatch sequence for that block slot.

Enforcing the condition that the same number of outputs must be produced for every execution of the block simplifies the block completion protocol at the RTs and the DTs. Each RT knows the exact number of tokens it must receive before it can signal completion. Likewise, DTs use the store mask known at block fetch

time and the DSN to identify store completion. Without a statically deterministic number of expected output tokens—as with early exits in hyperblocks or absence of null writes—the completion protocol must detect when no more outputs can be produced. This condition is complex to evaluate in the presence of predication, as it requires the polling of all ETs to identify if the execution of an output-producing instruction is nullified. Implicit predication exacerbates the problem as it requires a mechanism to examine the dataflow antecedents of an instruction to determine if it will ever execute and produce an output.

Protocol Latencies

A control signal generated at the GT incurs multiple cycles of latency to propagate to all the tiles. Table 4.8 provides the minimum latencies for several block events. For example, the minimum latency for executing the last instruction assigned to the farthest ET (bottom right corner) is 20 cycles. If the result of this instruction is a register output, that result takes a minimum of four cycles to reach an RT. Notifying the completion of the register outputs to the GT consumes up to four additional cycles. Combining all events for a single block, the overall lifetime of block from start to deallocation could involve a significant delay for control signal propagation. Hiding this delay is important for attaining high throughput.

In the TRIPS prototype implementation, all control protocols are pipelined. Except for fetch, a new operation for other protocols can be started in every cycle. The fetch for a second block can be initiated eight cycles after the first block. The fetch of a new block following a flush can be started three cycles after the flush. Such pipelining amortizes the latency of the control protocols across multiple tiles and blocks. For example, while one tile is performing the flush of speculative blocks, another could perform the commit of the non-speculative block, and a third tile could simultaneously be dispatching the instructions of a new block. As we demonstrate

Dispatch of first instruction to nearest ET	4
Dispatch of last instruction to farthest ET	17
Execution of first instruction in nearest ET	7
Execution of last instruction in farthest ET	20
Commit in nearest RT/DT	20
Commit in farthest RT/DT	24
Deallocation	32

Table 4.8: Minimum latencies for a few block events. Latencies are measured in processor cycles from the start of the fetch protocol in the GT.

in Chapter 6, such overlap is instrumental in reducing the overhead of distributed block control and management.

4.4.3 Discussion

Control of the distributed hardware units and management of the common execution state will be an important design component in future microarchitectures. The TRIPS prototype processor implements fine-grained control using a set of simple master-slave protocols running atop multiple control networks. The GT (master) generates control signals and drives them, one hop per cycle, to the other tiles (slaves) using the control networks. The protocols are latency tolerant; control signal propagation through the microarchitecture for one operation can be fully overlapped with another operation.

In our initial revision of the design, we implemented all of the control protocols using two shared networks. However, we observed that the contention for the network among the different protocols wasted the execution bandwidth of the processor. In fact, it was not possible to match the peak execution rate of the processor. Consequently, in the final design we implemented each protocol with a separate network at the cost of additional wiring between the tiles. Since the protocols were well-defined and fairly simple, migrating to a new implementation did not involve a significant redesign effort. The new networks—GRN and GCN—increased

the control signal bit widths between adjacent tiles by 49 bits from the original 216 bits in the GDN and the GSN, as shown in Figure 4.4. The tiled nature of the microarchitecture and point-to-point interconnections enabled this design change, even during an advanced stage of development without much complexity.

4.5 Performance Validation

The TRIPS prototype implementation included a significant pre-silicon verification component, whose purpose was to identify and fix any bugs before tapeout. Implementation bugs may cause incorrect execution or poor performance. For example, a bug in the branch computation logic might lead the execution on an incorrect control path, resulting in incorrect execution. However, a bug in the branch predictor may result in unnecessary mis-speculations. It results in poor performance, but does not cause incorrect execution. The first category of bugs is resolved by functional validation, which verifies the correctness of computation. The second category is resolved by performance validation, which verifies the correctness of performance. This section describes the latter, as it relates to the understanding of the performance of the TRIPS prototype processor.

4.5.1 Validation Phases

We validated the performance of the RTL implementation over two phases. During the first phase, we identified common microarchitectural events whose latency or throughput is typically critical for performance. For example, the latency of an instruction cache miss is generally on the critical path of execution and affects performance. Any bug that manifests as increased latency for the I-cache miss will degrade performance further. To identify such bugs, we measured the latency or throughput for several events using specific targeted tests and compared them against the design specification. During the second phase, we used several microbenchmarks and corre-

lated the overall performance of the RTL implementation with a detailed, high-level performance model of the prototype processor. During both phases, we identified several differences between the specification and the implementation, and addressed as many critical ones as the project schedule would allow, or were worth the effort to fix. We present our experience and observations in the following subsections.

Latency/Throughput Verification

We first manually verified the latency or throughput of common microarchitectural events that are important for performance. Table 4.9 provides a subset of the more than 100 events that we verified. We crafted several small tests in the TRIPS assembly language to target each event and ran them using the Synopsys VCS simulator for the processor RTL called *proc-rtl*. We generated event dumps from the simulator and observed them using the Synopsys VirSim waveform viewer. By studying the waveforms, we measured the latency for the microarchitectural event under study. We also used various event counters built into the processor RTL to examine the throughput of the microarchitectural event under study.

We then identified the core microarchitectural components that are likely to have a major effect on performance and treated their performance validation as an exercise in functional validation. These components were the dependence predictor, the branch predictor, the instruction cache, and the data cache. The individual tile logic designers verified these components in isolation from the rest of the processor by applying randomly generated test vectors and comparing the outputs on a cycle-by-cycle basis with the outputs of a high-level simulator model for that component. Every discrepancy was treated as bug and resolved. The entire process was repeated several times until no bugs were discovered. This exercise uncovered a notable bug in the replacement logic for the instruction cache. The bug caused an incorrect implementation of the LRU replacement policy, which did not

Event measured	Observation
Latency	
Network latency	one cycle per hop
Start of a block fetch	3 cycles after an I-cache fill
Dispatch of first register operand	5 cycles after block fetch
Execution of first instruction	7 cycles after block fetch
Commit of a block	20 cycles after block fetch
Deallocation of a block	32 cycles after block fetch
Load-to-use latency	5 cycles for cache hits
Store completion notification	5 cycles after block fetch
Register completion notification	18 cycles after block fetch
Store commit latency	8 cycles
Register commit latency	8 cycles
Execution of dependent instructions	back-to-back cycles (in same tile)
Execution of dependent instructions	two cycles apart (in neighboring tiles)
Throughput	
Peak block fetch rate	one every eight cycles
Peak block commit rate	one every eight cycles
Peak execution rate	16 instructions per cycle
Peak operand delivery rate (per tile)	one operand per direction per cycle (4)
Peak load rate	one per tile per cycle (4)
Peak store rate	one per tile per cycle (4)
Peak register rate	one read per tile per cycle (4)
Peak register rate	one write per tile per cycle (4)

Table 4.9: Microarchitectural events whose latencies or throughput were verified.

violate correct functional execution, but caused unnecessary I-cache misses.

In addition to the random tests, we devised special tests to verify the intended functionality of selected components such as the next-block predictor and the dependence predictor under known conditions. For example, in the case of the next-block predictor, we crafted tests with different correlation patterns to stress specific internal components such as the global predictor or the local predictor. We then ran these tests on *proc-rtl* and verified that the prediction accuracies were within expected bounds. We conducted similar targeted experiments for the dependence predictor. Neither of these exercises uncovered any additional performance bugs,

as the major bugs were discovered and resolved in the earlier functional verification steps.

Execution Cycles Verification

In this phase, we correlated the overall performance of the RTL implementation with a detailed performance model for the prototype processor. We developed two custom tools, *tsim-proc* and *tsim-critical*, to help in the performance correlation. The *tsim-proc* tool is a high-level, detailed performance simulator for the prototype processor. The *tsim-critical* tool observes various microarchitectural events that happen during the execution and computes the critical path, which is defined as the longest path of execution through the program. It attributes every cycle spent on the critical path to one of several types of dependences—data dependences in the program or microarchitectural dependences such as functional unit contention, operand routing network contention, and branch mispredictions. We describe both of these tools in greater depth in Chapter 5.

We formed a benchmark suite consisting of small kernels, which were drawn from the inner loops of various SPEC CPU2000 and signal processing workloads[1]. We crafted the suite such that each benchmark is a complete program and would execute within an acceptable time on *proc-rtl*. We ran these programs on *proc-rtl* and measured the execution cycles using performance counters architected into the processor design. We compared the cycles against the results obtained from *tsim-proc*. To normalize the effects of the memory system, we outfitted both simulators with identical, perfect secondary caches. In addition, we crafted the benchmarks such that their instruction and data were both cache resident.

Table 4.10 presents the results of the comparison on various kernels. The second column lists the execution cycles measured on *proc-rtl*. The next two columns

[1]Xia Chen and Robert McDonald formed this benchmark suite.

Benchmark	Execution time *tsim-proc* (cycles)	% difference *proc-rtl* (before)	% difference *proc-rtl* (after)
dhry	118808	-33	-10
ammp_1_hand	118278	-17	-4
fft4	3591	-14	-5
dct8x8	48891	-11	-11
fft2_GMTI_hand	101196	-11	-6
vadd_hand	92812	-11	-4
matrix_1	23231	-10	-6
sieve_hand	126162	-10	-1
transpose_GMTI_hand	71517	-10	-6
bzip2_3	107897	-9	-1
gzip_2	81340	-9	0
gzip_1	31347	-8	-7
gzip_2_hand	29753	-8	-3
sieve	94325	-8	-4
vadd_hand_tasl	87250	-8	-6
bzip2_1	121915	-7	1
equake_1	82872	-7	-2
fft2_GMTI	92031	-7	-6
fft4_GMTI	99547	-7	-5
parser_1	99405	-7	-13
transpose_GMTI	72117	-7	-4
art_3	91544	-6	-5
bzip2_2	80967	-6	1
twolf_3	100140	-6	2
vadd	129085	-6	-6
ammp_1	48442	-5	-5
doppler_GMTI	97194	-5	-8
fft4_GMTI_hand	54433	-5	-4
art_2	81139	-4	2
doppler_GMTI_hand	86930	-4	0
twolf_3_hand	48556	-4	1
ammp_2	104848	-3	-4
matrix_1_hand	45229	-2	-9
parser_1_hand	39384	-2	0
bzip2_1_hand	70193	0	11
art_1	103762	4	5
bzip2_3_hand	71162	9	5
MEAN (only under-estimates)	—	-8	-4

Table 4.10: Percentage difference in execution cycles between *tsim-proc* and *proc-rtl*. Negative numbers indicate that *proc-rtl* reports worse performance than *tsim-proc*.

IT pipeline bubbles during instruction cache refills
ET pipeline bubbles during instruction decode
Longer latency for block deallocation in the GT
Longer register commit latency in the RT
Longer block flush penalty
Incorrect prioritization of branch predictor operations
Operands to local and remote targets not sent in the same cycle

Table 4.11: Performance issues found and fixed in the RTL.

present the percentage differences in the execution cycles observed with *proc-rtl* and *tsim-proc*, before and after performance correlation. Positive numbers indicate that *proc-rtl* reports better performance than *tsim-proc*, whereas negative numbers indicate the opposite. As the results indicate, prior to any correlation, *proc-rtl* generally exhibited a worse performance than *tsim-proc*. The difference amounts to 8% on average, and as much as 33% in one benchmark.

We selected every benchmark that exhibited a negative difference of more than 5% as candidates for further examination. To identify the implementation features that contributed to the differences, we modified *tsim-critical* to observe the microarchitectural events in both *tsim-proc* and *proc-rtl* and compute the critical execution paths for each. By examining the critical paths in a fine-grained detail, we isolated many differences between *tsim-proc* and *proc-rtl*. Table 4.11 presents the most significant differences, and those that were worth the design effort and complexity to address in the implementation. After suitable design modifications, we observed that the execution cycles from *proc-rtl* and *tsim-proc* match within 4% on average. The number of benchmarks that exhibited a negative difference of more than 5% dropped from 25 to 12. In a few benchmarks such as *bzip2_1_hand*, the differences actually increased. We attribute this behavior to the differences in the store/load dependence prediction at the DT, which is sensitive to the precise arrival order of the store and any conflicting load operations at the DT, especially

during a small window of execution cycles. We suspect that the differences in the microarchitecture cause a different arrival order, forcing a shift in the dependence prediction to conservative, which lowers performance, or a shift to aggressive, which can performance.

4.5.2 Discussion

We developed the high-level performance model, *tsim-proc*, in conjunction with RTL design specification, not only to evaluate the performance of the prototype architecture, but also to drive RTL design decisions. As the design matured, suitable changes were made in the RTL either to accommodate area and timing constraints, or to reduce complexity. These conscious design changes and other inadvertent changes that slipped into the design contributed to the performance differences between *tsim-proc* and *proc-rtl*. We chose not to develop a performance model that is 100% cycle-accurate with respect to *proc-rtl*. While not infeasible, such a simulator would have been too slow for useful performance evaluation and taken a considerable effort that pushed the project schedule beyond acceptable limits.

4.6 Summary

In this chapter, we described the implementation of the TRIPS prototype chip. We described the heterogeneous processor tiles and the point-to-point networks that connect them. Our design, implementation, and verification effort was considerably simplified by this modular organization. We then described the control logic implementation and the protocols that manage the distributed computation in the processor. We showed how wires are treated as first-class constraints throughout the implementation and how different protocols are designed to recognize and tolerate the latency of signal propagation through the microarchitecture. Finally, we described the performance verification of the TRIPS prototype processor. We de-

scribed the verification of the latency and throughput of various microarchitectural events in the processor and how we correlated the performance of the prototype implementation with a high-level performance model. We observed that the individual protocols to fetch a block, begin execution, and complete execution themselves incur significant latency. But as our results in subsequent chapters show, this latency is mostly off the execution critical path, since multiple blocks overlap their operations with one another.

Chapter 5

Evaluation Methodology

The previous chapter described the ISA and the distributed microarchitecture of the TRIPS prototype processor. We use the same processor as the platform for our evaluation. Since the ISA and microarchitecture are quite different from conventional architectures, the evaluation requires the development of an entire software toolchain, including simulators and performance models, from the ground up, and there is little that can be leveraged off prior work. In particular, a detailed evaluation requires a suite of benchmarks, a compiler that produces aggressively optimized code, a simulator that models the microarchitecture faithfully, and finally, a set of analysis tools that offer insight into the performance bottlenecks in the architecture.

This chapter provides an overview of the various components required for performance evaluation. Section 5.1 describes the various workloads that comprise our evaluation suite. Section 5.2 and Section 5.3 describe the compilation infrastructure and where code quality was inadequate, the hand-optimizations that we applied to improve the performance of a benchmark. Section 5.4 describes the performance simulators that model the TRIPS architecture and our methodology for measuring the performance of a benchmark. Finally, Section 5.5 describes a critical path-based methodology to identify the performance bottlenecks of the TRIPS

Microbenchmarks		
dct8x8	2-D discrete cosine transform	data parallel
matrix	10x10 integer matrix multiplication	data parallel
sha	NIST secure hash algorithm	low ILP
vadd	1024-element floating-point vector addition	data parallel
LL Kernels		
conv	Time domain implementation of a FIR filter	data parallel
ct	Matrix transposition	data parallel
genalg	Genetic algorithm solving an optimization problem	control bound
EEMBC		
a2time01	Angle to time conversion	control bound
basefp01	Basic integer and floating point math	compute bound
rspeed01	Road speed calculation	control bound
tblook01	Table lookup and bilinear interpolation	control bound
bezier02	Bezier curve calculation	compute bound
autocor00	Finite length fixed-point autocorrelation	data parallel

Table 5.1: List of hand-optimized benchmarks used for evaluation.

architecture. It describes the critical path model and the algorithms that we developed to compute the critical path of execution efficiently and measure the effect of various microarchitectural events on performance.

5.1 Benchmarks

The TRIPS architecture is well suited for exploiting different kinds of parallelism—ILP, DLP, and TLP. In this dissertation, we focus exclusively on ILP and the performance of single-threaded workloads[1]. Accordingly, we draw workloads from a variety of sources—EEMBC, which is an industry-standard embedded processor workload suite [1], signal processing kernels from MIT Lincoln Labs, SPEC CPU2000 integer and floating point workloads [71], and a few custom microbenchmarks. Table 5.1 and Table 5.2 provide a listing of these benchmarks, their sources, and a brief description for each.

[1]For a detailed evaluation of the how the TRIPS architecture exploits DLP and TLP, we refer the reader to Sankaralingam's dissertation [132].

SPEC CPU Integer	
164.gzip	Compression
181.mcf	Combinatorial optimization
186.crafty	Game playing: chess
197.parser	Word processing
255.vortex	Object-oriented database
256.bzip2	Compression
300.twolf	Place and route simulator
SPEC CPU Floating Point	
168.wupwise	Physics/Quantum chromodynamics
171.swim	Shallow water modeling
172.mgrid	Multi-grid solver: 3D potential field
173.applu	Parabolic/Elliptic differential equations
177.mesa	3-D Graphics library
179.art	Image recognition / neural networks
200.sixtrack	High energy nuclear physics accelerator design
301.apsi	Meteorology: pollutant distribution

Table 5.2: List of SPEC benchmarks used for evaluation.

5.2 Compilation

We compile each workload using the TRIPS toolchain, which accepts C and FORTRAN programs and produces binaries that will execute on the TRIPS hardware. First, the compiler performs traditional scalar optimizations such as partial redundancy elimination, sparse conditional constant propagation, dead variable elimination, and array access strength reduction. It also performs various high-level transformations such as loop invariant code motion, loop flattening, and loop unrolling. It then generates TRIPS instructions and forms blocks using loop peeling, additional unrolling, if-conversion, and tail duplication, followed by various predication optimizations. Finally, it performs register allocation, peephole optimizations, instruction placement, and assembles the final object code.

The TRIPS development compiler currently lacks a few optimizations such as better fanout reduction and load/store dependence elimination that are essential for producing high quality TRIPS programs. Prior to the complete implementation of these optimizations, the performance of compiled binaries on the TRIPS

microarchitecture will be inadequate. To evaluate an upper-bound on achievable performance from the compiler, we hand-optimized the important kernels in several benchmarks[2]. We generated high-level TRIPS assembly called TIL (TRIPS Intermediate Language [148]) using the compiler and optimized the TIL files manually. We then fed the resultant files back into the TRIPS toolchain to perform instruction placement and produce the TRIPS object code. All benchmarks in Table 5.1 were hand-optimized, and unless otherwise noted, we report performance only for the hand-optimized versions.

5.3 Hand Optimization

The optimizations applied by hand include the following: a) better instruction merging, b) fanout reduction using predicate combining and ϕ-merging, c) better hyperblock formation, and d) load/store dependence elimination through better register allocation. In general, we observed that any optimization that increased the block size with useful instructions improved the overall performance. This result is not surprising, as large blocks amortize the overheads of distributed execution in the TRIPS microarchitecture. In this section, we highlight a few specific hand-optimizations that improved code quality significantly.

5.3.1 Instruction Merging

Figure 5.1 shows a kernel snippet extracted from *genalg*, which is one of the benchmarks in our evaluation suite. The top portion of the figure depicts the C source code. The second portion shows the equivalent TIL instructions for one iteration of the loop body, not including the register read and write instructions and necessary test instructions outside the loop iteration. The annotated comments for the TIL

[2]The benchmarks used in this dissertation were optimized by the author, Doug Burger, and Robert McDonald.

a) Loop extracted from genalg

```
for (x = c; rx > 0.0 && x < pop-1; x++, p_fitness++)
        rx -= *p_fitness;
```

b) Instruction sequence for the loop body

```
lw      $t4, ($t3) L[0]
fstod   $t5, $t4              ; fp single to double
fsub    $t6, $t0, $t5         ; rx  -= *p_fitness
addi    $t7, $t1, 1           ; x++
addi    $t8, $t3, 4           ; p_fitness++
fgt     $t9, $t6, $t100       ; rx > 0.0
tlt_t<$t9>    $t10, $t7, $t2  ; x < pop - 1
bro_t<$t10>   genalg$4        ; loop back
bro_f<$t9>    genalg$5        ; loop exit
bro_f<$t10>   genalg$5        ; loop exit
```

c) Predicate guards for live-outs

```
mov_f<$t9>   $rx, $t6
mov_f<$t10>  $rx, $t6
mov_f<$t9>   $x, $t7
mov_f<$t10>  $x, $t7
mov_f<$t9>   $pf, $t8
mov_f<$t10>  $pf, $t8
```

d) After disjoint instruction merging

```
bro_f<$t9, $t10>    genalg$5
mov_f<$t9, $t10>    $rx, $t6
mov_f<$t9, $t10>    $x, $t7
mov_f<$t9, $t10>    $pf, $t8
```

e) After predicate combining

```
movi_f<$t9, $t10>   $temp, 1

bro_f<$t9, $t10>    genalg$5
mov_t<$temp>        $rx, $t6
mov_t<$temp>        $x, $t7
mov_t<$temp>        $pf, $t8
```

Figure 5.1: Instruction merging in *genalg*.

instructions represent the corresponding C statements. The test instructions (`fgt` and `tlt`) represent the predicate-AND chain required for implementing the short-circuiting loop condition checks. The six move instructions in Figure 5.1c represent the definitions of three live registers `x`, `rx`, and `p_fitness` on the two loop exit paths.

If the loop executes for several iterations, statically unrolling it to fill an 128-instruction block with as many iterations as possible maximizes parallelism. This potentially high degree of unrolling exposes many opportunities for optimization. For example, the two exiting branch instructions, `bro_f<$t9>` and `bro_f<$t10>`, shown in Figure 5.1b are predicated on disjoint predicates—`$t9` and `$t10`. At most one of the predicates will be false at runtime. If `$t9` is false, the execution of the `tlt_t` instruction is inhibited, and no value will be produced for `$t10`. Otherwise, the `tlt_t` instruction will execute and produce a true or false value for

`$t10` depending on the test condition. Since only one, if any, of the two predicates will be false at runtime, the two branch instructions can be merged into a single branch instruction—`bro_f<$t9, $t10>`—as shown in Figure 5.1d. This optimization reduces one instruction from the block and creates room in the block for other instructions. Likewise, each pair of move instructions for the live registers shown in Figure 5.1c can also be reduced to one instruction—an elimination of three `mov` instructions overall. This optimization is typically known as *instruction merging* [92, 147], and is one of the optimization phases currently under development in the TRIPS compiler.

5.3.2 Predicate Combining

The three move instructions and the branch instruction shown in Figure 5.1d are candidates for further optimizations. Note that both of the loop exit predicates—`t9` and `t10`—must fan out to at least four consumers, the three move instructions and one branch. Without loss of generality, let us assume that each instruction can encode at most two targets. With this assumption, a total of four fanout instructions would be needed for the fanout of the two predicates. Instead the compiler can combine the two predicates with a special `movi` instruction as shown in Figure 5.1e and send the resulting predicate to the register defining instructions. This optimization reduces the fanout requirements—no fanout instructions are required for `t9` and `t10`, and one fanout instruction is required for `temp`. We call this optimization *predicate combining*.

Predicate combining adds an extra instruction, but reduces the overall fanout in the block. It also increases the dependence computation height for a few instructions, because the predicates must be combined first before they can sent to the consumers. But this optimization creates room in the block for the inclusion of additional unrolled iterations. In our experience, we observed that even at the expense

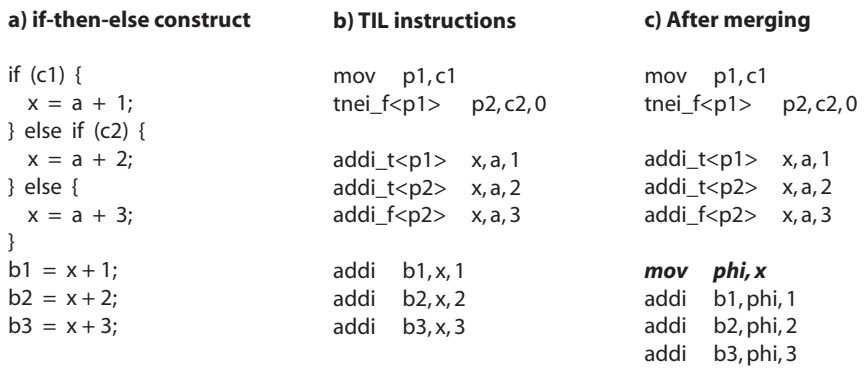


Figure 5.2: Example of ϕ-merging.

of dependence height extensions, predicate combining to fill a block maximally almost always provided performance improvements. In fact, it is the largest source of improvement in *genalg*, where the hand-optimized version performs better than the best compiled version by over 2.8×.

5.3.3 ϕ-merging

ϕ-merging is an optimization similar to predicate combining. It merges mutually exclusive definitions with a single ϕ-instruction to reduce the fanout. For example, consider the code snippet shown in Figure 5.2. The first portion shows a cascaded if-then-else construct defining the same variable `x` on three mutually exclusive paths. The second portion depicts the equivalent TIL instructions. In the absence of any optimizations, each of the three definitions of the variable `x` must target all of the consuming `addi` instructions, which requires separate fanout trees, although only one value will defined at runtime. Instead, as shown in Figure 5.2c, the mutually exclusive definitions can be combined with one ϕ-instruction, represented by the `mov` instruction. The ϕ-instruction can then target all the consumers with a fanout tree.

This optimization reduces the fanout communication for values defined on mutually exclusive paths. As before, it elongates the dependence computation height, but reduces the overall number of instructions in the block and enables the

unrolling more iterations of a loop in the same block. We found this optimization to be generally helpful in the SPECint benchmark *mcf*.

5.3.4 Other Optimizations

The optimizations described in the previous sections are enabling techniques for loop unrolling and generally provided better performance. We also applied other optimizations for improving performance: a) better register allocation of constants and global variables, b) re-organization of computation trees to speed up critical paths, c) elimination of redundant load instructions, d) register allocation to remove store-load dependences, and e) better hyperblock formation using control flow profiles. At the time of writing this dissertation, all of these optimization phases are under various stages of development in the TRIPS toolchain.

5.3.5 Performance Improvements

Table 5.3 compares the performance of the hand-optimized benchmarks with their corresponding compiled versions. We obtain these results using the TRIPS performance simulator, *tsim-proc*, which is described in Section 5.4. As shown in the table, hand optimizations produce improvements of up to 7× over compiled code and 2.8× on average. Most of these improvements result from a reduction in the block count, an attendant increase in the average block size, and a reduction in the overall instructions executed. The reduction in block count stems from better hyperblock formation enabled by fanout reduction optimizations. The reduction in instruction count stems from fanout reduction and better register allocation. We believe that a production quality compiler can automate the hand optimizations and as it matures, obtain performance similar to aggressively hand-optimized code.

BENCH	Inst. ratio (hand / tcc)	Block ratio (hand / tcc)	Block size hand (# insts)	Block size tcc (# insts)	Speedup (hand / tcc)
dct8x8	0.73	0.61	86.0	72.1	1.39
matrix	0.45	0.45	71.0	72.4	2.00
vadd	1.13	0.77	74.2	50.8	1.25
conv	0.84	0.81	87.0	84.7	1.34
ct	0.38	0.23	88.3	53.8	2.48
genalg	0.21	0.14	43.9	28.9	5.61
a2time01	0.70	0.37	73.2	38.4	2.92
autocor00	0.68	0.75	55.6	61.3	1.35
basefp01	0.56	0.24	96.9	42.1	6.98
bezier02	0.82	0.27	66.6	22.1	2.72
rspeed01	0.47	0.24	51.6	25.9	4.28
tblook01	0.80	0.59	72.3	52.9	1.58
MEAN	0.65	0.46	72.2	50.5	2.83

Table 5.3: Comparison of hand-optimized benchmarks with their compiled versions. The hand-optimized results are denoted by "hand" and the compiled versions are denoted by "tcc". Speedup is measured by comparing the execution cycles.

5.4 Simulators

The previous chapters and sections referred to a detailed performance model for the TRIPS microarchitecture called *tsim-proc*. This section provides additional details on the simulator and our overall methodology for simulation and performance measurements.

5.4.1 TRIPS Simulation

The simulator *tsim-proc* is a cycle-level, execution-driven simulator for the TRIPS prototype microarchitecture. It faithfully models all components of the prototype processor depicted in Table 4.2 and Figure 4.3. It models the tiles, network links, and the pipelines within each tile in great detail, simulates execution down mis-speculated paths, and reports aggregate performance statistics such as execution cycles, branch prediction accuracy, and cache miss rates. As reported in Chapter 4, on a test suite of small microbenchmarks whose footprint mostly fit in the L1 cache, *tsim-proc* is accurate to within 4% of the RTL-level processor simulator. We drive

Scenario	Expected latency range (min–max) in cycles
L1 hit	5–17
L2 hit	20–55
L2 miss	88–125

Table 5.4: Expected load-to-use latency for different scenarios in the TRIPS hardware. Latency is measured in processor cycles and corresponds to accesses to the memory address space hosted by TRIPS chip.

most of our evaluation in this dissertation using *tsim-proc* and various performance analysis tools based on *tsim-proc*.

The simulator, however, uses a different memory model than the TRIPS hardware. Table 5.4 provides the expected hardware latency from execution of a load at an ET to the receipt of the load value at the consuming ET. For each scenario, the table provides a range of expected latencies in the absence of contention on the networks and tiles along the path from the ET executing the load to the ET consuming the load. A range of latencies exist for each scenario as the relative distances between the source ET and the DT that contains the address, the DT and the L2 cache bank that contains the address, the L2 cache bank and the SDRAM controller, and finally, the DT and the consuming ET can all vary. The simulator models the latency of a L1 hit faithfully. However, it does not model the NUCA L2 cache or the bus connection with main memory. Instead, it models a flat, 1 MB L2 cache with a 12-cycle hit latency. This assumption corresponds to the average expected access latency in the NUCA L2 cache and the average load-to-use latency for a L2 hit shown in Table 5.4. Since most of our experiments measure only the performance of the processor microarchitecture, the simulations assume perfect caching in the L2, resulting in no L2 misses.

The simulator *tsim-proc* is quite slow—it simulates 1500 cycles per second on average—and is not flexible enough for design space exploration. For this exer-

cise, we developed another simulator called *tsim-flex*, which models the prototype processor at a much higher level of detail than *tsim-proc* and simulates an order of magnitude faster. It models an identical L2 memory system as *tsim-proc* and a main memory whose access latency is 150 cycles. Compared to the expected load-to-use hardware latencies shown in Table 5.4, *tsim-flex* models identical L1 hit behavior, the average case L2 hit behavior, and a higher penalty for a L2 miss. Under the same set of assumptions used for validating *tsim-proc*, we validated *tsim-flex* to be accurate within 12% of the RTL-level simulator.

5.4.2 Alpha Simulation

To compare the performance of the TRIPS architecture against a conventional processor architecture, we use the Alpha EV6 (21264) processor. We use the Alpha EV6 because it uses a microarchitecture that is tuned for aggressive ILP and fast clock rates. In addition, it is supported by an industry-leading compiler that produces aggressively optimized code for an ISA that lends itself to efficient execution. We compile various workloads on the Compaq workstation using the native Gem compiler with the optimization flags "-O4 -arch ev6". To normalize the effects of the memory system and operating system, we measure the Alpha performance using a detailed simulator called *sim-alpha*, which has been validated against real hardware and found to be accurate within 2% on a set of microbenchmarks and 18% on a set of SPEC CPU2000 workloads [43]. Table 5.5 shows the parameters used in the simulations measuring Alpha performance.

5.4.3 Reducing Simulation Time

The detailed microarchitectural execution of an entire workload using a software simulator is a time-consuming affair. It is often four to five orders of magnitude slower than real hardware, even on the fastest desktop machines available today.

Issue of width of six instructions (4 integer and 2 FP) 20-entry integer issue queue and a 15-entry FP issue queue 80-entry ROB Four integer units, two pipelined FP units
1-cycle integer ALU, 7-cycle integer multiply 4-cycle FP add/multiple/load 12-15 cycle native FP divide 18-33 cycle native FP sqrt
64KB, 2-way set associative I-cache, 64-byte cache blocks 64KB, 2-way set associative D-cache, 64-byte cache blocks 1-cycle I-cache access on correct set prediction 3-cycle D-cache access 32-entry load queue, 32-entry store queue, 8-entry MSHR
Tournament branch predictor Two-level local predictor with 10-bit, 1024-entry L1 and 1024-entry, 3-bit L2 counters 4096-entry global predictor with 2-bit counters 4096-entry choice predictor with 2-bit counters
Perfect L2 cache, flat latency of 12 cycles

Table 5.5: Simulator parameters for Alpha 21264.

Simulating a distributed microarchitecture such as TRIPS, with its many interacting components, compounds this slowdown further and is prohibitively expensive for large workloads such as SPEC. To keep the simulation times tractable, we resort to simulating only a small portion of an entire workload. The following paragraphs describe the methodology that we used to select the portions to simulate for various workloads.

Microbenchmarks, LL Kernels, and EEMBC

These workloads are crafted such that an inner kernel performing the desired computation iterates multiple times over the same data. For each workload, we select an input data set and an iteration count such that the kernel computation dominates the overall execution.

SPEC CPU2000

We use the *ref* input data set for simulating these workloads. To reduce the simulation time, we select one representative region in each workload using the *SimPoint* methodology [144] and simulate that region. To compare identical program regions across different compilations and architectures, we use the following methodology.

First, we select a single 100-million instruction, early SimPoint region computed for the Alpha instruction set. On a few benchmarks—*mcf*, *sixtrack*, and *wupwise*— where the early SimPoint regions are farther into the program, and hence prohibitively expensive to simulate, we select one early region among multiple SimPoint regions. We use the same regions previously identified and published by Sherwood et al. [144, 145].

Next, we stretch the identified region to the boundaries of a function call or a return. In the absence of inlining of the bounding functions, this step ensures that any comparison between different architectures or between different compilations for the same architecture is performed on identical program regions. Table 5.6 shows these regions for the SPEC workloads compatible with our methodology. The table shows the input data set used for simulation and the start and end of the simulated region, which are specified in terms of the function call instances called or returned from. For example, the table shows that for the benchmark *mcf*, the simulated region starts at the 1631^{st} invocation of the function `refresh_potential` and ends at the return from the 1702^{nd} invocation of the same function. However, in *swim* the simulated region starts at the call of the first of the invocation of `calc2_` and ends at the return of the same invocation. By starting and stopping simulation at the boundaries of a non-inlined function, we simulate identical program regions.

Discussion: The TRIPS performance simulators—*tsim-proc* and *tsim-flex*— provide coarse-grained performance profiles such as execution cycles, cache misses,

Benchmark	Input Arguments	Start Function	End Function
164.gzip	input.graphic 60	fill_window : 32	fill_window : 44
181.mcf	inp.in	refresh_potential : 1631	refresh_potential : 1702
186.crafty	crafty.in	main : 1	OutputMove : 62
197.parser	2.1.dict -batch	prepare_to_parse : 31	parse : 31
255.vortex	bendian2.raw	BMT_QueryOn : 1040	BMT_QueryOn : 1106
256.bzip2	input.program 58	generateMTFValues : 2	generateMTFValues : 2
300.twolf	ref	ucxx2 : 607430	ucxx2 : 631211
168.wupwise	wupwise.in	main : 1	dlaran_ : 1012001
171.swim	swim.in	calc2_ : 1	calc2_ : 1
172.mgrid	mgrid.in	resid_ : 14	resid_ : 14
173.applu	applu.in	blts_ : 3	blts_ : 3
177.mesa	-frame 1000 -meshfile mesa.in -ppmfile	g1_write_texture_span : 4769641	g1_write_texture_span : 4832551
179.art	mesa.ppm -scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 200 -objects 10	match : 1	match : 2
200.sixtrack	inp.in	umlauf_ : 1012	umlauf_ : 1037
301.apsi	apsi.in	dftdx_ : 2	dtfdx_ : 27

Table 5.6: SimPoint regions for SPEC CPU2000 workloads. The number next to ":" indicates the particular instance of the function call.

and prediction accuracies. While they are good indicators of performance, they are not adequate for a more fine-grained evaluation of the bottlenecks in the architecture. The active instruction window in the TRIPS processor could feature thousands of microarchitectural events, some of which occur on concurrent paths, while others are dependent on each other. Naturally, some events affect the overall execution time more than others. Understanding the interactions among these events can help a designer overcome various bottlenecks and improve performance. In the remainder of this chapter, we describe our methodology for analyzing the bottlenecks in the TRIPS architecture.

5.5 Critical Path Analysis

Critical path analysis has been proven as an effective technique for understanding the bottlenecks of an architecture [53]. This analysis abstracts the execution of a program with a directed acyclic graph constructed using a simulator or a runtime profiler. Nodes in the graph represent microarchitectural events that occur during the lifetime of the program, while edges represent the dependence constraints among the events. These constraints include both data dependences among the instructions and machine constraints specific to the architecture. Different insights can be gained by analyzing the dependence graph. For example, one can identify if long execution times are a result of poor instruction-level parallelism (ILP) in a program. If the critical path—defined as the longest path in the graph—consists of a large fraction of data dependence edges in the program, then it is because of low available ILP. A different composition of the critical path may indicate other constraints. An architect can thus obtain the relative critical path contribution of each type of dependence constraint and identify the potential bottlenecks among them.

The complexity of computing the critical path depends on the instruction window size of the processor. The TRIPS prototype processor with its 16-wide issue, 1024-entry instruction window, and distributed microarchitecture increases the complexity considerably. The complexity also depends on the number of different types of dependence constraints and the granularities at which they are tracked by the graph. In the simplest form, only aggregate critical path contributions of a constraint may need to be tracked, for example, the number of data cache miss cycles that appear on the critical path. However, for a better bottleneck analysis one may need to track these constraints at a finer-grained level, for example, which data cache bank, which program block, or which load instruction contributed the most cache miss cycles on the critical path. These different levels of granularity have a multiplicative effect on the amount of in-flight state required for analysis and

increase the complexity accordingly.

In this section, we describe our extensions to the simulation-based critical path framework—previously developed for conventional processors—to analyze the performance of the TRIPS architecture [107]. We develop an algorithm to manage the large in-flight state required for critical path analysis efficiently. The algorithm trades off the simulator memory required for maintaining the dependence graph with the cost of traversing the graph. We show how a careful tradeoff can reduce the complexity of computing the critical path significantly. This section presents details of the TRIPS critical path framework and describes the algorithms for computing the critical path. It also presents experimental results that demonstrate how different algorithms perform with respect to the computation time required for critical path analysis. The next chapter presents results obtained using the framework and identifies the performance bottlenecks in the TRIPS architecture.

5.5.1 Prior Critical Path Models

The notion of critical path models for processor architectures is not new. Prior research has focused on one of the following: identifying criticality of specific classes of instructions, critical path modeling, critical path prediction and optimizations to improve performance. Early research on critical path analysis has generally focused on understanding the performance of load instructions. Srinivasan et al. quantify a measure of load criticality called latency tolerance [155]. In subsequent work, Srinivasan et al. propose a heuristics-based hardware predictor of critical loads and quantitatively compare criticality-based prefetching techniques with locality-based techniques [154]. Other researchers have also proposed hardware estimators of critical loads and provided techniques to improve cache performance of critical loads at the expense of non-critical ones [58, 124]. All of these approaches are specific to load instructions and cannot be easily extended to other instructions or microarchi-

tectural resources.

Our work is closest to the dependence graph-based models of the critical path developed by Fields et al. [53] and Tune et al. [171]. As explained before, these models abstract the execution of a program with a dependence graph. By manipulating the graph in different ways, they show how different measures of criticality—slack [52], tautness [171] and interaction costs [54]—can be computed efficiently with a simulator. They use these models to develop runtime critical path predictors. In a later work, Fields et al. show how the model can be used to gain insights about secondary critical paths [54]. Our research extends these models for the TRIPS architecture.

Researchers also developed techniques to predict the criticality of all types of instructions [28, 53, 138, 170]. These techniques provide critical path measures to varying degrees of accuracy and have been applied to improve value prediction performance, clustered architecture scheduling, and power consumption.

5.5.2 TRIPS Critical Path Model

The critical path model for the TRIPS architecture is derived from the dependence-graph model previously developed for superscalar architectures [53]. The model represents various microarchitectural events as nodes in a directed acyclic graph. Edges between the nodes represent dependence constraints among the events. Figure 5.3 shows a typical dependence graph constructed for a slice of four blocks seen during the program execution. In addition to representing the usual constraints such as data dependences, branch mispredictions, and finite instruction window sizes, the TRIPS critical path model also represents constraints imposed by block-atomic execution and operand routing.

Each node in the graph maintains the following information.

- Type of the event: block fetch, operand communication, register read, etc.

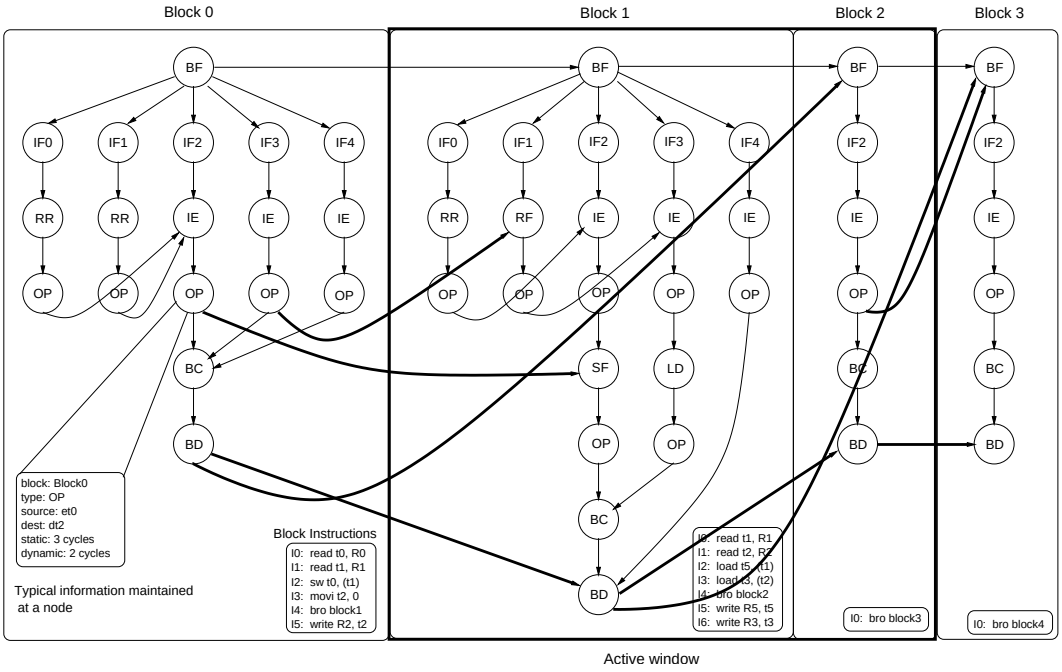


Figure 5.3: Critical path model for the TRIPS architecture. The example shows the dependence graph for four blocks and a machine window size of two blocks. Bold arrows depict inter-block dependences.

- Static delay: statically determinable cycles consumed by an event, e.g., latency of a integer multiply.
- Dynamic delay: latencies introduced by dynamic events, e.g., execution stall cycles due to contention for the issue slot.
- Information about the block and/or instruction associated with the event.
- Information about the tile or network link where the event occurs.

Edges in the graph do not maintain any special information. Instead they represent various constraints of the execution model. The block-atomic execution model relies on a few global tasks that are performed on behalf of an entire block. These tasks introduce dependence constraints for operations not only within the

Name	Events	Dependences	Dependence Edge
BF	Fetch of a block	In-order block fetches Recovery from control misprediction Finite window	$BF_{i-1} \rightarrow BF_i$ $OP \rightarrow BF$ $BD_{i-w} \rightarrow BF_i$
IF	Instruction Fetch	Must follow block fetch	$BF \rightarrow IF$
RR	Read of a register	Must follow read instruction fetch	$IF \rightarrow RR$
IE	Instruction execute	Must follow instruction fetch Can execute only after operands have been received	$IF \rightarrow IE$ $OP \rightarrow IE$
OP	Operand communication	Can communicate result only after execution has completed Can communicate register values after register read	$IE \rightarrow OP$ $RR \rightarrow OP$
BC	Block execution completion	Block completes after all outputs have been produced	$OP \rightarrow BC$
BD	Block commit	Block commits after it completes Blocks must begin their commit operations in-order	$BC \rightarrow BD$ $BD_{i-1} \rightarrow BD_i$
RF	Register forward	Register forwarded after value is produced Register forwarded after read instruction is fetched	$OP \rightarrow RF$ $IF \rightarrow RF$
LD	Load reply	Load reply happens after address is fetched	$OP \rightarrow LD$ $LD \rightarrow OP$
SF	Store forward	Forward happens after value is received	$OP \rightarrow SF$ $SF \rightarrow OP$

Table 5.7: Dependences for the TRIPS critical path model.

block and but also in other blocks. We summarize these constraints in the following paragraphs and in Table 5.7.

Intra-Block Fetch Dependences: The control logic at the GT determines the address of the next block to fetch. This event is represented by the graph node BF and denotes the availability of the block's instructions in the cache and the start of the fetch process. The GT takes at least eight cycles to initiate the fetch for the entire block. These eight cycles are recorded as the static latency of the BF event. Any additional latency, for example, due to cache misses, is recorded as the dynamic latency of the event.

The delivery of instructions from the I-cache banks to the respective ETs is represented by the IF nodes. The distributed nature of the fetch is represented by the independent IF nodes that are all only dependent on the global block fetch event ($BF \rightarrow IF$). Each IF event has a statically determinable latency based on the location of the tile to which the instruction is dispatched.

Intra-Block Execution Dependences: An instruction executes when all of its operands are ready. Some of these operands may be block inputs that are read from the register banks and delivered on the operand network to the execution tiles. The register read event (RR) represents the read of a block input value from the register bank. It is dependent on the fetch of the corresponding read instruction ($IF \rightarrow RR$). The operand transfer event (OP) represents routing of the value to the tile containing the consumer instruction. The latency for operand routing includes both statically determinable cycles computed from the number of routing hops and additional cycles resulting from contention for the intermediate network links. The event IE represents the execution of an instruction. After execution, the instruction may route the result to a dependent instruction in the same block. Such data dependences manifest as edges from the execution events of producers to the execution events of consumers via operand communication events ($IE \rightarrow OP \rightarrow IE$).

Intra-Block Commit Dependences: A block completes its execution when all of its outputs—registers, stores and branch target—have been computed. Edges from instruction execution events to the block completion event (BC) via operand communication events represent this dependence ($IE \rightarrow OP \rightarrow BC$). Once a block is known to have completed, the outputs can be committed. This constraint is denoted by the dependence $BC \rightarrow BD$.

Inter-Block Dependences: The fetch of a block can proceed only after the fetch for the previous block has started. This in-order block fetch dependence is represented by the ($BF_{i-1} \rightarrow BF_i$) edges. Similarly, blocks can complete their commit operations only in order. This constraint is represented by $BD_{i-1} \rightarrow BD_i$ edges. Instructions across different blocks could have data dependences through registers. The hardware has the capability to forward register values from producer instructions in a block to consumers in another block, without waiting for the previous block to commit. This forwarding event is represented by the graph node RF, and associated intra-block fetch dependence $IF \rightarrow RF$ and inter-block dependence $OP \rightarrow RF$.

The hardware supports the execution of up to eight blocks in flight. The fetch of a block can thus proceed only after the deallocation of the eighth block preceding the current one. This dependence is represented by the $BD_{i-w} \rightarrow BF_i$ edges, where w denotes the window size in blocks. Figure 5.3 depicts a window size of 2 blocks. Finally, branch misprediction in a block constrains the fetch of the successor block. Once the branch instruction is executed and the target communicated to the GT, the fetch process can be initiated. The sequence of dependence edges $IE \rightarrow OP \rightarrow BF$ represents this constraint.

Store-Load Dependences: Load instructions compute the effective addresses at an execution tile and sends them on the network to data tiles. Data tiles read the value for loads from the cache and route them back to consumer instructions. The cache access is represented by the event LD. Hit latencies appear as static delays and miss latencies appear as dynamic delays. The associated dependences for this sequence of events are represented by the edges $IE \rightarrow OP$, $OP \rightarrow LD$, and $LD \rightarrow OP$. Occasionally, a prior store in the same block or preceding block may have the same address as the load. The load can obtain the correct value only after the store has been received at the data tile. Once the store arrives, the load-store

queues at the data tile can forward the value from the store to the matching load. This forwarding event is represented by the node SF.

Analyzing the Dependence Graph

The dependence graph of a program's execution can offer key insights into the various bottlenecks for performance. By analyzing the graph in various ways, we can compute different performance metrics and determine the degree to which a certain architectural constraint or hardware resource affects performance.

Critical Path: The longest path in the dependence graph—measured by summing the weights of the nodes in the path—- from the BF event in the first block to the BD event in the last block provides the critical path of execution through the program. By examining the composition of the nodes along the path, one can summarize the contributions of each type of event, each tile or network link in the processor, each program block, or even each instruction in the program to the overall execution of the program. For example, one can determine that a significant fraction of the critical path cycles results from issue slot contention stalls at the tile ET0 while trying to execute the instruction at address `0xbadf00d0`. Such information can then be fed back to the compiler so that it can find a better placement for the instruction, perhaps by moving it to a different execution tile to eliminate the contention stall cycles. We note that there may be multiple paths from the first event to the last event in the program's dependence graph. If there are multiple candidates for the critical path, we pick only one of the paths.

Cost: The cost of an event is the reduction in the program's execution cycles if that event was idealized. The idealization of the event can be modeled by either eliminating its causal dependences or reducing its latency in the program dependence graph, as appropriate. A subsequent re-computation of the critical path provides

the cost of the event. For example, the cost of branch misprediction events can be computed by removing all edges corresponding to branch mispredictions from the dependence graph and recomputing the critical path. The cost of load misses in the primary cache can be evaluated by modifying the latency of load reply event to that of the primary cache latency. The difference in the critical path lengths between the unmodified and modified dependence graph is the cost of the event.

The cost of an event indicates the degree to which secondary critical paths and concurrent events affect performance. For example, consider a certain event that contributes 50% of the execution cycles to the critical path. While the event is certainly a bottleneck for performance, completely idealizing it may not provide a corresponding two-fold increase in performance. Other paths that were previously secondary may become critical and reveal additional bottlenecks. The next section describes our framework and the algorithms to compute the critical path composition and costs.

5.5.3 Critical Path Framework

The critical path framework for the TRIPS prototype processor consists of two major components: a) *tsim-proc* and b) a dependence graph constructor and critical path analyzer. We simulate programs compiled for the TRIPS architecture using *tsim-proc*. We use traces generated by the simulator to construct the dependence graph of execution. The critical path analyzer then traverses the dependence graph and outputs the critical path information at the desired level of granularity.

The critical path can be computed at different granularities.

- An event-level summary provides the number of cycles spent for each type of event on the critical path.
- A block-level summary provides the number of cycles for each event type in each program block executed on the critical path.

- A tile-level summary provides the contributions of each hardware tile and an instruction-level summary provides the contributions of each program instruction executed on the critical path.

We modified *tsim-proc* to output a trace of the various microarchitectural events that happened during the execution of a program. The trace contains details of each event such as the cycle when it occurred and information about the block or instruction(s) associated with it. We construct the dependence graph using the trace and compute critical paths using the algorithms described in the following sections.

Critical path analysis requires an effective management of the large dependence graph state. Three factors determine the complexity of the algorithm that computes the critical path: a) the size of the graph saved for analysis, b) the number of graph nodes visited during the analysis, and c) the granularity at which the critical path composition is computed. The first factor determines the memory requirements, while the other two determine the computational requirements of the algorithm. In this section, we review two traditional approaches that have opposing requirements on memory and computation. We then present a new algorithm that exploits certain properties of the dependence graph, lowers the requirements on both computation and memory, and delivers the best performance.

Backward-Pass Algorithm

This algorithm starts at the BD node for the last block. At each step of the algorithm, it visits another node by proceeding to the latest parent node that satisfied the current node's constraints. It terminates at the BF node for the first block. The sequence of the nodes visited is the critical path of execution and by aggregating various information at each of these nodes one can obtain the critical path summaries at different levels of granularity. The advantage of the algorithm is that it does not visit any node that is not on the critical path. However, it requires the

entire graph to be constructed and saved before the critical path can be computed. This requirement is clearly intractable for large programs.

Forward-Pass Algorithm

Prior work on critical path analysis used a simple forward-pass algorithm and saved only a portion of the graph at any given time [53, 171]. The key property of the graph exploited by this algorithm is the fact that no dependence constraint can span more blocks beyond that allowed by the maximum window size of the machine. Consequently, this algorithm maintains only the sub-graph of events for a window of $w + 1$ blocks at any given time, where w is the maximum window size. However, each node must maintain summaries of the critical path in reaching that node.

The critical path summary at each node contains the number of cycles spent for every type of microarchitectural event on the critical path leading to that node. Consequently, the cost of copying the summary from one node to another is proportional to the number of different types of events tracked by the tool. The granularity of the critical path composition determines the cost of computing the summaries. If a block-level granularity is desired, the summaries should include the number of critical path cycles for every event in every block. Consequently, the copying costs are proportional to the product of both the number of different blocks executed in the program and the number of different types of events. A tile-level or a instruction-level breakdown has a similar multiplicative effect on the cost of computing the critical path summaries.

The algorithm starts by constructing the graph for the first $w + 1$ blocks. For every node in the first block, it visits all of its successors. During each visit, it propagates all critical path information tracked thus far at a node to the successor. The successor updates its information only if the parent satisfied its constraints the latest. It then adds a new block ($w + 2$, in sequence) to the graph, removes the

sub-graph corresponding to the first, and repeats the process for the second block.

This algorithm reduces the memory requirements dramatically. However it visits every node in the program's overall graph. In addition, during each visit, a node must copy the complete critical path breakdowns to its successor. Depending on the required granularity of the breakdowns, these copying costs grow proportionately and for large programs can be prohibitively expensive.

Mixed Algorithm

The backward-pass algorithm requires copies only along the critical path, but its memory requirements are intractable. By contrast, the forward-pass algorithm keeps only a small sub-graph in memory, but since it visits every node, its copy requirements can be intractable. A desirable algorithm is one that does not require the entire graph and at the same time does not visit every node to compute the critical path.

A key property of the dependence graph is that for the sub-graph corresponding to an arbitrary window of contiguous blocks, the number of edges from nodes within the window to those outside can always be bounded. This property applies to the dependence graphs for both conventional superscalar and TRIPS architectures. For the TRIPS architecture, these out-going edges can source only a few "output" nodes: a) one BF node, enforcing in-order block fetch start events, b) one BD node, enforcing in-order block commit events, c) one branch communication node for any branch mispredictions, d) one or more register output communication nodes, and e) one or more store communication nodes. The latter two set of nodes can be bounded as they can only belong to the most recently seen eight blocks, each of which can have only up to 32 register writes and 32 stores as permitted by the ISA. We exploit this property in composing an algorithm that uses a combination of both backward and forward passes. One can extend the algorithm fairly easily

for a conventional superscalar architecture.

The algorithm maintains the sub-graph of events for a sliding window of $r+8$ blocks, where r is a large number such that the graph can be feasibly accommodated in memory. The algorithm starts by constructing the graph for the first $r+8$ blocks. It then does a backward pass starting from each "output" node (in blocks $r-7$ to r) and collects the critical path information at these nodes. For each output node, it then propagates the critical path information to all its successors similar to the forward pass algorithm. It then removes the top r blocks and adds the next r blocks to the graph. The whole process continues until the critical path information is collected at the commit node for the last block.

Depending on the value for r, the algorithm reduces the number of graph nodes visited and consequently, the number of times the critical path information is copied from one node to another. A large value for r imposes a greater memory requirement for maintaining the in-flight graph state compared to the forward-pass algorithm. But it amortizes that cost by visiting only those nodes that are on the critical path leading to an output node. In our experiments, we found that best setting for r was one that consumed most of the available memory. Note that if r is set to 1, the algorithm is similar to the forward-pass algorithm described above and if set to ∞, it defaults to the backward-pass algorithm.

Computing Costs

The algorithms described in the previous sections compute the composition of the critical path at the desired level of granularity. All of them can be easily extended to compute the cost of a specific event.

Forward Pass: Recall that each step in the forward pass algorithm involves a propagation phase, in which it aggregates the delays of two nodes. To compute the cost of an event, the algorithm discards the delay of that event. In addition, if nec-

essary, it avoids propagating the accumulated delays along the edges corresponding to the event. In our implementation, the algorithm disregards only the edges representing branch mispredictions and finite window stall events. For all other events, it propagates the delays along their causal dependence edges, but sets their latencies to zero.

Backward Pass: To compute the cost of an event, the backward pass algorithm first creates a new copy of the entire dependence graph. It then removes from the new graph the edges corresponding to the event, if necessary, and sets the delay for the event's node to zero. Finally, it performs a backward pass on the new graph to measure its critical path length. The difference in the critical path lengths between the unmodified and the modified dependence graphs provides the cost of the event.

Mixed: The mixed pass algorithm creates a new copy of the in-memory sub-graph and modifies it suitably. It performs a backward pass starting at each output node and computes the cost. It then propagates the costs from the output node of the current sub-graph to the input nodes of the next sub-graph similar to the forward pass algorithm.

5.5.4 Results

This section shows the results of the critical path analysis on a select set of benchmarks. Our primary goal in this section is to illustrate the runtime complexity of the different algorithms for critical path computation and not to demonstrate the performance bottlenecks in the architecture. Consequently we limit ourselves to a set of five benchmarks from Table 5.1—*a2time01, bezier02, dct8x8, sha, matrix*. These benchmarks are iterative, repetitive, and have a small enough working set that fit in the level one caches. Table 5.8 provides a listing of the benchmarks along with the number of blocks and instructions encountered during dynamic execution

Name	Block counts	Instruction counts	Execution Time (cycles)
a2time01	16880	112402	477212
bezier02	461694	3807281	2984977
dct8x8	40194	3614106	196342
matrix	25624	1355074	230833
sha	15576	1252784	582178

Table 5.8: Benchmark set used for evaluating critical path algorithms.

and the observed execution time of these programs during the evaluation.

5.5.5 Algorithm Performance

We first demonstrate the performance of the mixed forward-backward pass algorithm. Figure 5.4 shows the speed of the critical path framework for different region sizes—the parameter r that determines the number of blocks for which the tool maintains the graph in memory. The x-axis varies the region size and the y-axis shows the analysis time measured in seconds. For these results, the tool computes the critical path at a block-level granularity. For each sample point in the graph, we perform a number of experiments on a dedicated desktop machine and report the average analysis time. The labels next to the points for the benchmark *sha* in the bottom graph represent the peak memory consumed during the critical path analysis.

Across all benchmarks, the analysis times improve dramatically as we increase the region sizes from 8 blocks to 64 blocks, at which point the benefits of further increases begin to taper off. At smaller region sizes, the cost of maintaining the graph in memory is insignificant compared to the cost of copying the critical path summaries across different nodes. Higher region sizes increase the memory requirements, but decrease the copying costs. We observe minor improvements for region sizes up to 512 blocks (256 for the benchmark *sha*). Beyond this size, the memory requirements of the algorithm exceed the capacity of the host machine (1 GB) and

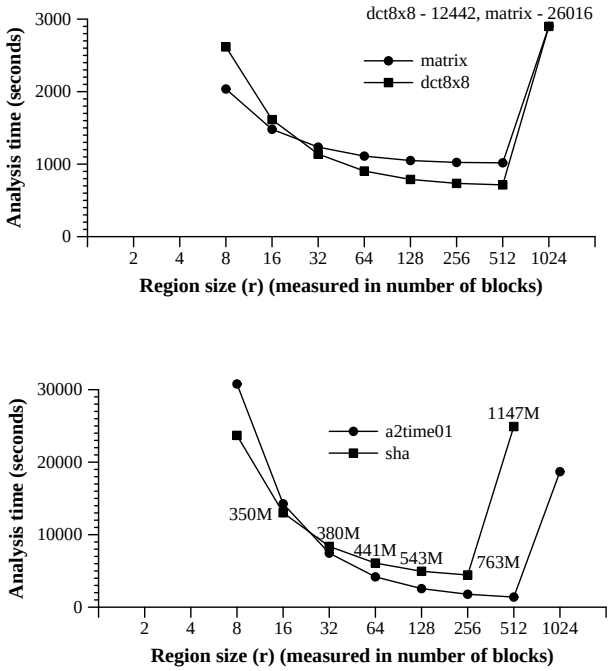


Figure 5.4: Sensitivity of analysis time to region sizes.

the resultant disk swap activity causes a precipitous slowdown in the speed of analysis. For example, in *sha* the memory requirements gradually increase up to 763 MB for a region size of 256. For higher region sizes, the memory requirements increase to 1147 MB, which exceeds the capacity of the host machine and affects the analysis time negatively.

Different benchmarks exhibit different speedups in analysis time. This is because the cost of block-level breakdowns in the critical path summaries is proportional to the number of different program blocks encountered during the execution. Benchmarks *matrix* and *dct8x8* contain fewer blocks than benchmarks *a2time01* and *sha*. Consequently they exhibit relatively modest improvements of (2×-3×) when varying the region sizes from 8 to 512. On the other hand, the benchmark *a2time01* exhibits nearly 30× improvement over the same range.

These results show that when computing rich critical path information, the

Granularity	a2time01	bezier02	dct8x8	matrix	sha
event-level	2.27	1.87	3.30	3.73	3.80
block-level	4.02	3.67	4.63	5.11	10.93
tile-level	2.75	2.51	4.22	4.71	7.33
instruction-level	2.90	2.01	8.65	9.56	8.77
tsim-proc speed (cycles/s)	1359	1494	1258	1149	1420

Table 5.9: Relative slowdown of analysis at varying levels of granularity.

mixed algorithm can deliver orders of magnitude improvements in performance with favorable region sizes. The best performing algorithm is one that just saturates the memory capacity of the host machine.

5.5.6 Speed of the Critical Path Framework

The speed of the overall critical path framework depends on three components: a) the speed of the cycle-level simulator, b) the granularity of the computed critical summaries, and c) the speed of algorithm computing the critical path. Table 5.9 compares the overhead of computing the critical path composition at different granularities with the baseline cycle-level simulator—*tsim-proc*. For every benchmark, it shows the simulation speed measured in simulated cycles per seconds for *tsim-proc* and the relative slowdown of the critical path analysis at four levels of granularity. For this study, we used the mixed forward-backward pass algorithm with a region size of 256.

Computing event-level breakdowns causes the baseline cycle-level simulation to slow down by a factor of 1.8×–3.8× across different benchmarks. Adding block-level breakdowns to the analysis causes additional slowdowns of 1.4×–3.8×. The differences in the benchmarks arise from the number of different blocks simulated during the execution. Computing the tile-level breakdowns, however, causes a fairly uniform slowdown of about 20%–30% compared to event-level breakdowns. This result is because during the backward pass, the constant number of tiles cause a

Event	BC	BD	BF	IE	IF	LD	OP	RR	RF
% cycles	1.6	4.6	7.0	24.6	8.1	2.6	38.0	6.4	7.2

Table 5.10: Critical path breakdown for *matrix*. Numbers indicate the percentage cycles on the critical path.

uniform amount of state to be copied from one node to another. The last row shows the cost of computing the contributions of each individual instruction in the most critical program block. The analysis is faster compared to the the block-level analysis for benchmarks *a2time01* and *bezier01*. This result is because these benchmarks have more program blocks than they have instructions in the most critical block, whereas the opposite is true for benchmarks *dct8x8* and *matrix*. The instruction-level analysis tracks only instructions of a single block and hence, proceeds faster than the block-level analysis.

As shown in Table 5.9, the speed of critical path analysis can be reduced dramatically depending on the desired granularity of the computation. To keep the analysis tractable, a designer ought to perform critical path analysis, starting with an event-level view and progressively add finer granularities for select portions of the program or hardware resources. For example, if a designer can identify the most critical blocks, s/he can obtain additional information just for that set of blocks with a different simulation.

5.5.7 Discussion

The critical path information for a program can be used in a number of ways described below.

Bottleneck Analysis: An event-level breakdown can provide the bottlenecks for performance. For example, consider the critical path breakdown for the execution of the benchmark *matrix* shown in Table 5.10. It shows that nearly 38% of the

Block Name	(Static,	Dynamic)	Total Delay (cycles)
matrix_mult$2	(74352,	56216)	130568
matrix_check$1	(26047,	31440)	57487
matrix_mult$1	(23969,	13785)	37754
main$4	(1812,	236)	2048
matrix_check	(1802,	239)	2041

a) Block-level breakdown for the program matrix

Instruction	(Static,	Dynamic)	Total Delay
addi	(3496,	4428)	7924
mul	(3285,	1965)	5250
mul	(2380,	1777)	4157
add	(1985,	777)	2762
tlti_f	(1552,	886)	2438

b) Instruction-level breakdown for block matrix_mult$2 in matrix

Figure 5.5: Detailed critical path breakdown for *matrix*.

execution time is spent in operand communication. The static component of the operand communication latency corresponds to the number of hops. The dynamic component results from network link contention. Each of these components can be reduced with improved placements for the critical instructions. By obtaining a block-level or instruction-level breakdown for the critical path, one can identify the program blocks and instructions contributing to the critical latencies and focus the scheduling policies towards them. Figure 5.5 shows an example of these breakdowns for the benchmark *matrix*. It shows that the most critical block in the program is `matrix_mult$2`, followed by `matrix_check$1`, and the most critical instruction within the block `matrix_mult$2` is the `addi` instruction. We present the results obtained using the analysis for other benchmarks in Chapter 6.

Performance Validation: As described in Chapter 4, we used critical path breakdowns to correlate the performance of *tsim-proc* with the RTL implementation. We applied the critical path analysis on the microarchitectural events observed with both simulators and computed the critical path breakdowns. By computing

the breakdowns at different granularities, we pin-pointed the discrepancies to specific tiles, program blocks, or program instructions. This ability eased the effort in performance correlation considerably.

Instruction Scheduling: The TRIPS toolchain has used critical path analysis to improve instruction scheduling [38]. One of the instruction scheduling algorithms in the compiler is simulated annealing, which attempts to arrive at an optimal schedule by randomly perturbing the schedule during each step. It is predominantly used for computing a performance upper-bound for evaluating the quality of the compiler-generated schedules. Instead of using random perturbations during each annealing step, a guided annealer focuses on the critical instructions. This optimization results in faster converge times, offering a two-fold speedup in some cases [38].

Hand Optimization: Finally, we have used critical path analysis in directing hand-optimizations towards critical program regions. In Appendix B, we describe one case study of hand-optimization using the guidance from the critical path analysis.

5.6 Summary

In this chapter, we presented the overall methodology for evaluating various features of the TRIPS architecture. We described the suite of benchmarks, the compilation infrastructure, and the set of hand-optimizations applied to produce benchmark code of high quality. We showed that the hand-optimizations out-perform the best-compiled versions of various benchmarks by a factor of 2.8 on average. We described the details of the performance simulators that model the TRIPS architecture in detail. Finally, we described in depth the critical path framework used for identifying various performance bottlenecks in the architecture. We described the algorithms

for determining the critical path and showed experimental results that demonstrate that an efficient management of simulation state can provide an order of magnitude improvement in the simulation time required for a detailed critical path analysis.

Chapter 6

Experimental Results

The previous chapter laid the experimental framework for evaluating the TRIPS architecture. This chapter presents the results of our evaluation. The goals for the evaluation are multi-fold: (a) measure the performance of the TRIPS architecture and compare it to other architectures, (b) measure the potential for performance improvements, (c) identify architectural and microarchitectural bottlenecks that constrain performance, and (d) investigate techniques that mitigate the effect of various bottlenecks and improve performance.

We organize this chapter as follows. Section 6.1 reports the performance of the TRIPS prototype architecture and a comparison with the Alpha 21264 architecture. Section 6.2 examines the parallelism that exists in various workloads and shows how constraints such as L2 cache misses, limited issue width, and limited instruction window capacity affect performance. The remainder of the chapter presents the results of a detailed critical path analysis. We identify performance bottlenecks in instruction supply, data supply, distributed microarchitecture, and the dataflow ISA. We also outline architectural and microarchitectural techniques that can potentially alleviate the effect of a few bottlenecks.

Benchmark	Performance speedup
conv	2.22
ct	2.27
dct8x8	2.68
matrix	3.63
sha	0.92
vadd	1.83
a2time01	4.50
autocor00	2.18
bezier02	4.63
rspeed01	4.92
MEAN	2.98

Table 6.1: Performance speedup of the TRIPS hardware over Alpha 21264. Results were obtained using hardware performance counters for TRIPS and *sim-alpha* for the Alpha 21264.

6.1 Performance of the TRIPS Architecture

This section reports the raw performance of the TRIPS architecture and compares it with the Alpha 21264 architecture. We first run several workloads on the TRIPS prototype hardware and measure their performance using hardware performance counters. While these counters report aggregate performance of the hardware, they do not monitor performance at a fine-grained instruction-level granularity. Therefore we turn to various tools derived from *tsim-proc* to understand the performance of the TRIPS architecture in greater detail—in terms of its instruction throughput and the degree to which it exploits a large instruction window for parallelism.

6.1.1 TRIPS Hardware Results

Table 6.1 presents the speedup of the TRIPS hardware over the Alpha 21264 microarchitecture. We use hardware performance counters to measure the performance of the TRIPS hardware and *sim-alpha* to measure the performance of the Alpha 21264 microarchitecture. For a fair comparison between a hardware platform and

a simulator for another microarchitecture, we normalize as many features as possible, and where impossible, use simulation parameters that favor the Alpha 21264. Specifically, we normalize system call effects so that neither the TRIPS hardware nor the Alpha 21264 counts the execution time spent in servicing system calls in a program. In addition, we normalize the memory system by configuring *sim-alpha* to use a perfect secondary cache. The TRIPS hardware, however, incurs the latencies of misses in the secondary cache and accesses to main memory, as depicted in Table 5.4. We leave the rest of the microarchitectural parameters in *sim-alpha* identical to the Alpha 21264 hardware as presented in Table 5.5 in the previous chapter.

Table 6.1 illustrates that the TRIPS hardware obtains significant speedup over the Alpha 21264 microarchitecture. The speedup is measured by computing the ratios of the execution cycles observed on each platform. In these measurements, we assume that both the platforms can be clocked at equivalent frequencies[1]. This speedup ranges from 0.9× to 4.9×, and averages 3× on the set of hand-optimized benchmarks in our evaluation suite. We omit three benchmarks—*genalg*, *basefp*, and *tblook*—from the TRIPS hardware measurements, as their hand-optimizations contain `fdiv` instructions, which are not supported by the TRIPS hardware. Generally, the speedups obtained by TRIPS are due to its higher execution bandwidth and a larger window of instructions from which the microarchitecture exploits parallelism. TRIPS also possesses exactly twice the data cache bandwidth as the Alpha 21264. The higher bandwidth yields nearly double the speedup in *vadd*, whose performance is predominantly dictated by the available memory bandwidth. The benchmark *sha* exhibits a slowdown in TRIPS, because it is dominated by a few long dependent chains. The available parallelism is mined effectively by the Alpha, whereas the TRIPS processor incurs overheads of distributed execution, which inhibits exploitation of the low available parallelism in *sha*.

[1]Such an assumption is not unrealistic as a custom design of the TRIPS chip can support high clock rates competitive with the Alpha at equivalent process technologies.

BENCH	Alpha-21264 IPC	TRIPS IPC	Instruction ratio (TRIPS / Alpha)
dct8x8	1.69	4.87	1.01
matrix	1.68	4.60	0.71
sha	2.28	2.10	1.00
vadd	3.03	6.51	1.11
conv	2.08	6.01	1.11
ct	2.31	5.17	0.59
genalg	1.05	1.55	1.13
a2time01	0.94	4.18	0.96
autocor00	1.96	3.81	0.87
basefp01	0.84	3.96	0.54
bezier02	1.06	4.18	0.94
rspeed01	1.03	3.31	0.79
tblook01	1.30	1.77	1.70
MEAN	1.63	4.00	0.95

Table 6.2: Comparison of instruction throughput with the Alpha 21264. Results were obtained by simulation—*tsim-proc* for TRIPS and *sim-alpha* for the Alpha 21264.

6.1.2 Instruction Throughput

Table 6.2 compares the instruction throughputs observed on Alpha 21264 and TRIPS. We obtained these results using simulation for both Alpha and TRIPS, as the TRIPS hardware does not count the instructions executed in the program. The second column shows the IPC obtained by a Alpha 21264 core, as measured using *sim-alpha*. The third column shows the IPC measured using *tsim-proc*. The last column shows the ratio of the instructions executed by the two architectures. We count only the actual instructions that are executed at run-time to compute the IPC. NOP instructions and instructions that receive non-matching predicates are not counted. To compare the capabilities of the microarchitectures directly, we configure both simulators to use identical secondary memory systems—perfect 1 MB L2 cache with a flat latency of 12 cycles.

As shown in the table, the Alpha and the TRIPS architectures use different number of instructions to complete the execution of a workload. In *basefp*, TRIPS allocates a lot of constant data to registers, whereas Alpha loads the constants

Benchmark	Alpha 21264 IPC	TRIPS IPC	Instruction ratio (TRIPS / Alpha)	TRIPS speedup
SPECint				
164.gzip	1.42	1.60	1.79	0.63
181.mcf	0.54	0.69	0.84	1.53
186.crafty	1.17	1.27	2.08	0.52
197.parser	1.18	1.13	1.38	0.70
255.vortex	1.21	0.73	2.64	0.23
256.bzip2	1.40	1.34	2.38	0.40
300.twolf	1.00	1.24	1.79	0.69
SPECfp				
168.wupwise	1.40	1.22	1.82	0.48
171.swim	2.29	6.79	1.59	1.85
172.mgrid	1.33	4.07	1.71	1.78
173.applu	0.89	2.99	6.64	0.50
177.mesa	1.10	1.81	1.96	0.84
179.art	0.95	2.35	3.05	0.81
200.sixtrack	1.37	1.35	2.32	0.42
301.apsi	1.26	3.40	2.65	1.02
MEAN	1.23	2.13	2.31	0.83

Table 6.3: Performance of SPEC workloads. Results were obtained by simulation—*tsim-proc* for TRIPS and *sim-alpha* for the Alpha 21264.

from memory. This optimization yields nearly 50% reduction in the number of instructions executed by the TRIPS processor. In *tblook*, the speculatively hoisted, but eventually nullified instructions contribute to more than 70% increase in the instructions executed. However, in many cases, TRIPS uses fewer instructions and yet, sustains greater IPC than the Alpha 21264. The IPCs obtained by the TRIPS processor range from 1.5 to 6.5 and average 4.0 on the hand-optimized suite of benchmarks. Comparatively, the Alpha 21264 sustains IPCs in the range 0.83 to 3.0 on the same set of benchmarks. These results are illustrative of both the availability of parallelism in the workloads and the ability of the TRIPS architecture to exploit greater parallelism.

Performance of SPEC Workloads

Table 6.3 compares the performance of the SPEC workloads running the *ref* input set on the TRIPS architecture and the Alpha 21264. We compiled the workloads using the best available compilers for both architectures and obtained these results using simulation. Once again, we configure the simulators for both architectures to use a perfect L2 cache with a 12-cycle access latency. We reduce simulation time by simulating only SimPoint regions as described in Chapter 5. The second and third columns depict the IPCs on the two architectures. The fourth column depicts the ratio of the number instructions to execute the same program region in the two architectures. The last column depicts the speedup obtained by the TRIPS architecture. These results indicate that except on four benchmarks—*mcf*, *swim*, *mgrid*, and *apsi*—the TRIPS architecture exhibits significant performance slowdown when compared to the Alpha 21264 architecture. On average its performance is 17% worse than the Alpha 21264. In general, the instruction throughputs (IPCs) are higher than Alpha, but TRIPS also executes far more instructions. For example, in the benchmark *applu*, TRIPS executes nearly 6.6× more instructions than the Alpha.

These results depict the TRIPS architecture unfavorably in comparison to Alpha. Much of the slowdown in performance stems from the increased instruction count in the TRIPS workloads. The additional instructions in TRIPS come from four sources: a) fanout instructions due to the restricted target encoding the ISA, b) sign extension instructions due to the lack of adequate instructions in the ISA for signed arithmetic, c) hoisted instructions on falsely predicated paths, and d) unnecessary load and store instructions generated by the TRIPS compiler. While some of the inefficiencies are artifacts of the ISA, others are artifacts of an unoptimized compiler. For example, most of the additional instructions in the benchmark *applu* arise from fanout instructions that can be optimized in the compiler and additional

State	Description
Wasted	
mis-speculated	no correct block is mapped to this slot, no program is being executed
nop	`NOP` instruction mapped to this slot
mispredicated	valid instruction mapped to this slot, but execution cancelled by non-matching predicate, or implicit predication
Useful	
committed	mapped block has completed execution, block waiting to be committed
slotted	block is mapped to this slot, but no instruction mapped yet
retired	instruction in slot completed execution, waiting for block to commit
waiting	valid instruction mapped to this slot, instruction waiting for an operand

Table 6.4: Different states of occupancy for an instruction window slot.

load/store instructions that can be eliminated using better register allocation or reuse of previously loaded values. We note that the TRIPS compiler is still under active development and as described in Chapter 5, several optimizations have not been implemented in the compiler completely. As the compiler matures, we expect the TRIPS results to improve and exhibit better performance.

6.1.3 Instruction Window Utilization

The TRIPS processor has a much larger instruction window than conventional superscalar processors to exploit parallelism. We now examine the degree to which the processor exploits this resource in practice. We use *tsim-proc* to track execution on a cycle-level basis and examine the occupancy of each slot in the instruction window every cycle. In any given cycle, a slot can be any of the following seven states: *waiting*, *retired*, *slotted*, *committed*, *mispredicated*, *nop*, or *mis-speculated*. Table 6.4 describes each of these states in detail.

A large instruction window is only as useful as the processor's ability to fill

it with *useful* instructions. Ideally, the TRIPS processor would fill its entire window with only useful instructions. This scenario requires all speculation to be correct and every block to both contain and execute all of its 128 instructions. In practice, control/data mis-speculations and instruction cache misses cause periods of poor occupancy in the instruction window. These factors contribute to the *mis-speculated* state described in Table 6.4. Furthermore, a block may waste some instruction window slots due to `NOP` instructions. The hardware allocates exactly 128 slots for executing every block, even if it has fewer than 128 instructions. The unused slots are padded with `NOP` instructions either statically by the compiler or dynamically by the hardware. These instructions contribute to the *nop* state represented in Table 6.4. Finally, instructions whose execution is inhibited because of predication also waste their slots. Recall from Chapter 3 that an instruction's execution may be inhibited directly by the receipt of a non-matching predicate or implicitly by the non-arrival of an operand. These conditions contribute to the *mispredicated* state. All three effects described in this paragraph reduce the effective occupancy of the instruction window and limit the scope of exploiting parallelism.

In addition to filling an instruction window with useful instructions, the hardware must also ensure that the window is not occupied by any block for an extended period of time. Extended occupancy by any single block eventually stalls the hardware from fetching and executing new blocks. In conventional processors, instruction window stalls are a common consequence of long latency to service data cache misses. Such stalls are common to the TRIPS processor as well. In addition, distributed execution results in additional cycles of occupancy. For example, even if the block has no register or store outputs, the block commit protocol must incur a minimum latency of eight cycles, which forces the block to occupy the window for the duration of that period. This state of occupancy, represented by *committed*, is unique to distributed execution. Similarly, the block fetch protocol must reserve a

Type	Description	Benchmark	Block size # dynamic insts	Prediction accuracy
LB_GP	large blocks, good control prediction	*basefp01*	96.9	99.7
SB_GP	small blocks, good control prediction	*rspeed01*	51.6	97.4
LB_PP	large blocks, poor control prediction	*dct8x8*	86.0	88.6
SB_PP	small blocks, poor control prediction	*genalg*	44.6	82.0

Table 6.5: Benchmark categories for evaluating window utilization.

slot for the block even before any instruction has been dispatched to the reservation stations. This state of occupancy, represented by *slotted* is also unique to distributed execution.

The remaining two states—*waiting* and *retired*—have analogies to conventional superscalar execution. The *retired* state is equivalent to a younger instruction completing execution and waiting for older instructions to commit. Similarly, the *waiting* state corresponds to instructions waiting to be woken by producer instructions. However, in contrast to a conventional processor, the TRIPS processor extends the duration of both these states due to the latency of operand communication and the distributed protocol for detecting the completion of block execution.

We illustrate the occupancy of the instruction window with four examples, one for each of the four categories described in Table 6.5. The categories are organized based on the average number of instructions executed in each block and the control flow prediction accuracy. The prefixes `LB` and `SB` denote large blocks and small blocks respectively, whereas the suffixes `GP` and `PP` denote good and poor control flow prediction. For example, the category labeled `SB_PP` denote benchmarks with both small blocks and poor control flow prediction. Figures 6.1– 6.4 depict the occupancy for the four categories over the course of execution. The x-axis represents different samples, each of which corresponds to a 1000-cycle or 10000-cycle execution period, as appropriate. For each sample, the y-axis charts the average occupancy of the instruction window during that period and shows a breakdown for the different states of occupancy. We configure *tsim-proc* as described in the Section 5.4, outfit

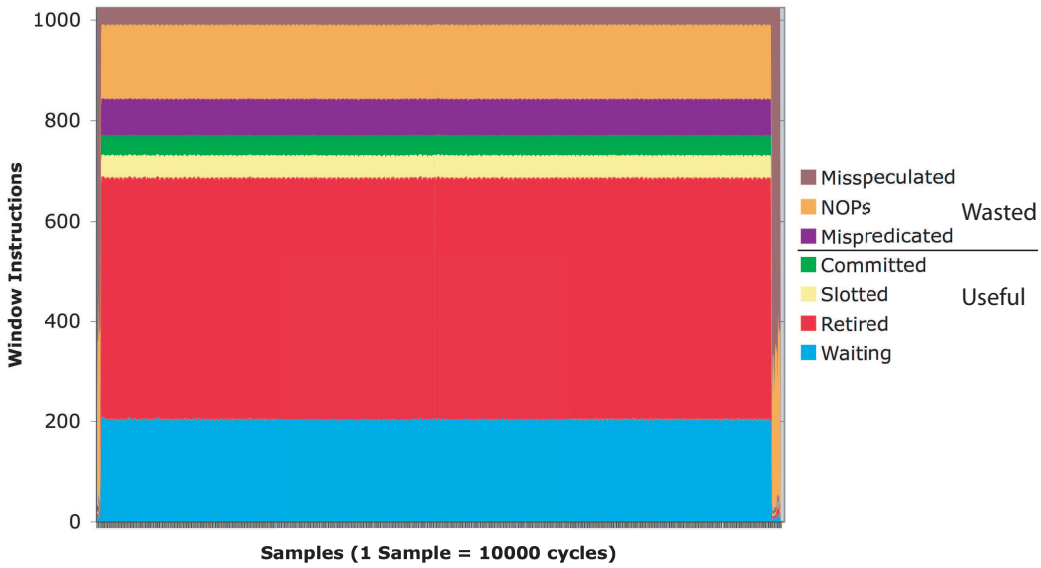


Figure 6.1: Window utilization for *basefp01*, LB_GP.

it with a perfect L2 cache as described in the previous sections, and monitor the window occupancy. We describe our observations in the following paragraphs.

LB_GP: As expected, this category yields the best occupancy. The near-perfect control predictor in *basefp* manages to fill the instruction window with eight blocks for a majority of the cycles. The eight 128-instruction blocks almost fill the 1024-entry instruction window. The occasional misprediction does result in periods of occupancy by mis-speculated blocks, which contribute to the small white section in the chart. The top two shades of gray correspond to `NOP` instructions and mis-predicated instructions. From Table 6.5 we observe that blocks in *basefp* have 97 executed instructions on average. These instructions summed across all in-flight blocks provide an effective window utilization of more than 75%, as illustrated by the bottom four categories.

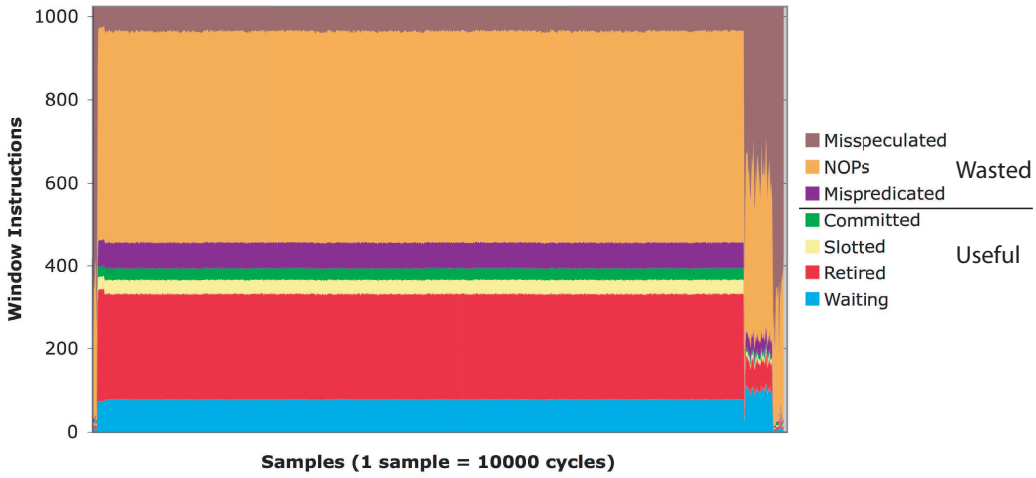


Figure 6.2: Window utilization for *rspeed01*, SB_GP.

SB_GP: The good control prediction accuracy in *rspeed* provides a window that is full with eight blocks most of the time. However, most blocks in *rspeed* are small—on average each block executes only 51 instructions. The majority of the instructions are `NOPs` and mispredicated instructions. These features reduce the effective window utilization to only 40% on average an shown in Figure 6.2. The fact that `NOP` instructions—depicted by the second section from the top—are more than half of all instructions in the window illustrates that the benchmark can benefit from the inclusion of more basic blocks within a single block.

LB_PP: In benchmark *dct8x8*, nearly 30% of the window is wasted by control mis-speculations, as illustrated by the white sections in Figure 6.3. This result is expected because the benchmark has a relatively high misprediction rate of 12%. The benchmark also contains several `NOP` instructions and suffers from fetch bandwidth limitations, which reduce the effective window utilization further.

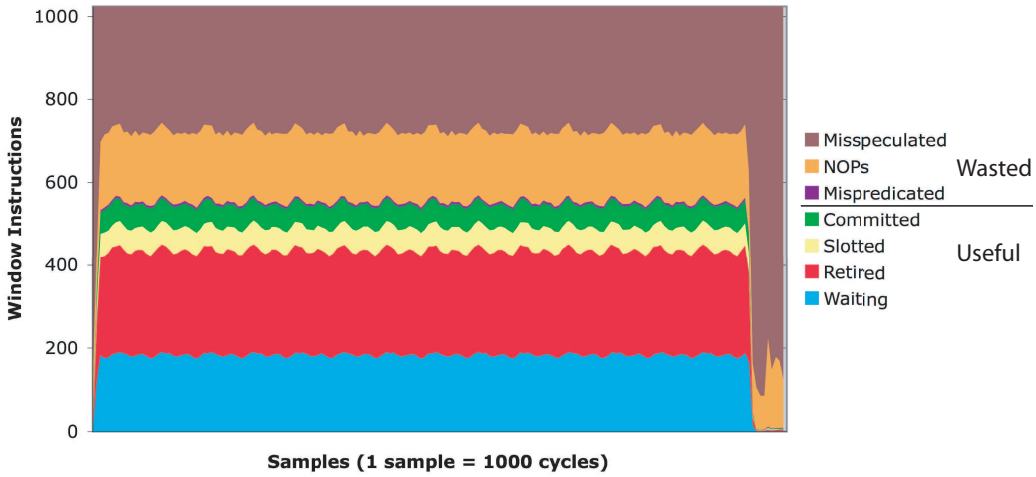


Figure 6.3: Window utilization for *dct8x8*, LB_PP.

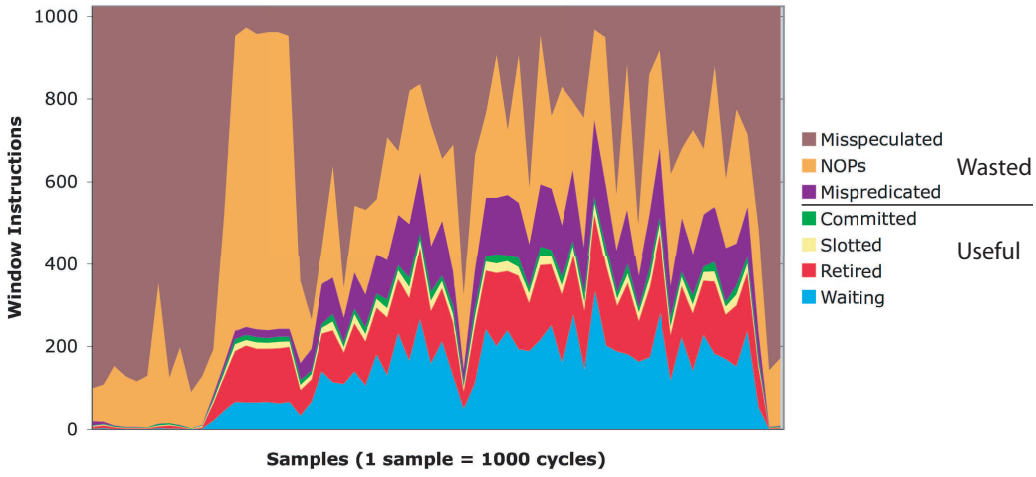


Figure 6.4: Window utilization for *genalg*, SB_PP.

SB_PP: This case yields the worst instruction window utilization. The benchmark *genalg* has a misprediction rate of 18%, and executes only 44 instructions on average. The low block size is a consequence of both predication and `NOPs`, as illustrated by the relative heights of the top two sections in Figure 6.4. Poor control prediction and I-cache misses, especially during the first trimester of execution, also contribute significantly to the poor utilization of the instruction window.

Summary

Across all workloads, the *waiting* and *retired* states account for most of the slot occupancy among the executed instructions. The next two states—*slotted* and *committed*, which represent the occupancy resulting from the distributed fetch and commit protocols are not nearly as prominent, indicating that they do not affect performance significantly. We re-visit these protocols in Section 6.3 and show quantitatively that it is indeed the case.

6.1.4 Discussion

The results from this section show that the TRIPS processor manages to utilize around 75% of the instruction window in the best case, and less than 30% in the worst case. However, this net utilization is larger than the maximum supported window size in conventional superscalar processors. Comparatively, among the conventional superscalar processors, POWER4 supports a maximum size of around 200 in-flight instructions. As our results in this section and subsequent sections illustrate, the TRIPS processor exploits this large window size and large execution bandwidth to sustain greater parallelism and performance than current processors.

6.2 TRIPS ILP Extraction

The TRIPS prototype processor is designed for a maximum execution rate of 16 instructions per cycle, yet it sustains much less in practice. The reasons for the less-than-ideal instruction throughput could be manifold, one of which is the availability of parallelism in the workloads. The hardware can mine the parallelism only if it exists in the application. Therefore, we must first evaluate the extent of available ILP in various workloads and compare it to the achieved ILP in the TRIPS processor. In this section, we present the results of that evaluation. We also measure the effect of a few common microarchitectural constraints such as issue width, instruction window capacities, and memory hierarchies on ILP exploited by the TRIPS processor.

6.2.1 Dataflow Limit

Prior research studies have explored the amount of ILP that exists in typical programs [13, 110, 129, 151, 166, 173]. There is no consensus on the exact ILP that is available in workloads. However, there is sufficient evidence to show that more ILP exists than can be harvested by a machine bounded by various constraints. In this evaluation, we compute an approximate bound for the amount of the available parallelism in each workload. We assume an ideal machine that has perfect caches, zero-cycle data communication latencies, infinite execution resources, and identical functional unit latencies as the baseline TRIPS processor. Furthermore, we assume that the machine obeys only the true data dependences that exist in the program. However, we do not eliminate any program stack dependences, and to keep our evaluation tractable we assume a finite, but large instruction window of 128 contiguous program blocks (up to 16K instructions) and a machine issue width of 64. We call the resulting instruction throughput the *ideal ILP* in the program. The true dataflow limit ILP will be higher if false dependences that exists among stack and data memory accesses can be removed, or if parallelism exists in

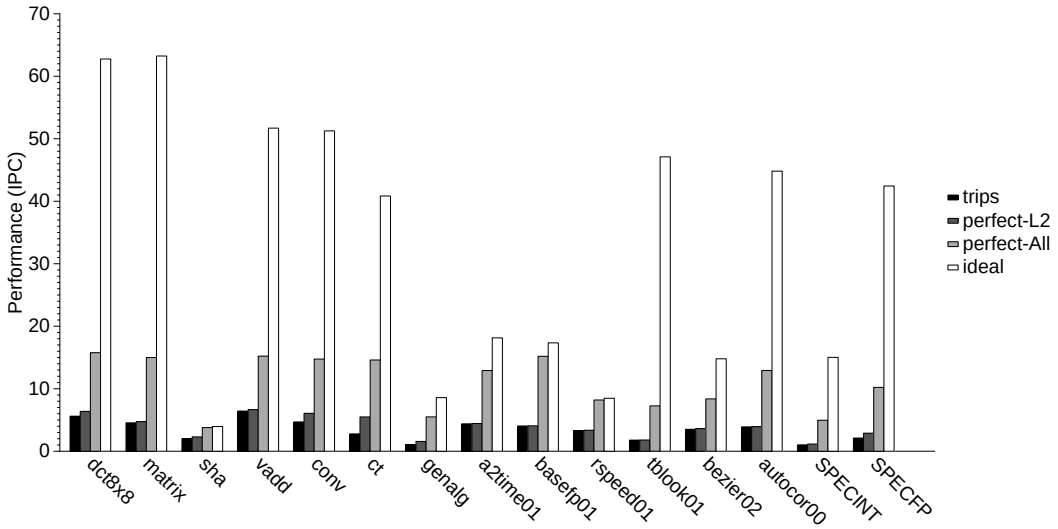


Figure 6.5: Available and observed ILP under various machine constraints. Results were obtained by simulation using *tsim-flex*.

far-flung program regions [13]. In the latter case, the available parallelism can be exploited by a machine with a very large instruction window—hundreds of thousands or millions of instructions deep—and issue width, or a machine that exploits ILP from non-contiguous program regions [15, 152]. Nevertheless, the ideal ILP metric is illustrative of the extent of available parallelism in a program and useful for comparing against the baseline TRIPS processor which also issues widely from a large contiguous program window.

Figure 6.5 presents the ILP that exists in different workloads and compares it with the obtained ILP under various conditions. These results were obtained using simulation with *tsim-flex*, which offers the ability to simulate perfect operating conditions for various microarchitectural features. For every benchmark, we depict four bars representing the obtained IPC at various conditions. The white bars, which represent the ideal IPC, indicate that benchmarks exhibit a wide range of parallelism, ranging from 4 to 64 instructions per cycle. Some benchmarks such as

dct8x8 and *matrix* are highly parallel attaining an ideal IPC of nearly 64, which is equal to the issue width of the ideal machine. However, as shown by the dark bars, the sustained IPC in the baseline TRIPS processor is significantly less. With the exception of *sha*, which is fundamentally limited due to the presence a small number of long dependence chains, the baseline processor harvests less than half the available ILP in all benchmarks, and often much less, averaging only 16% across the suite of benchmarks.

Implications: A considerable difference exists between available ILP and harvested ILP. The TRIPS processor possesses an instruction window that is 16× smaller and an issue width that is 4× smaller than the ideal machine. These capacity constraints affect the amount of parallelism extracted by the hardware. Incorrect speculation and imperfect caches also inhibit parallelism. In the following sections, we progressively add a few realistic constraints to the ideal machine and examine how they affect performance.

6.2.2 Effect of L2 misses

We measure the effect of misses in the L2 cache by idealizing its behavior. We assume that all L2 accesses hit in the cache and incur a hit latency of 12 cycles as described in Section 5.4. The second bar for every benchmark in Figure 6.5 presents the resulting performance. The results indicate that most hand-optimized benchmarks exhibit only minor improvements in performance, as their working sets largely fit in the cache. A notable exception is *ct*, which operates on streaming data and exhibits poor temporal locality. We observed that many SPECint benchmarks did not benefit significantly from a perfect L2 cache. The only exceptions are *mcf*, which exhibited more than a two-fold increase in performance, *parser*, which exhibited a 20% increase in performance, and *vortex*, which exhibited nearly 40% increase in performance. Many SPECfp benchmarks, however, exhibited marked increases in

performance due to perfect L2 caches. Averaged across the SPECfp suite, the improvements are 37%.

Implications: Memory latency affects the performance of certain benchmarks significantly. Across the suite of benchmarks, we observe a performance difference of nearly 18% between realistic and perfect L2 cache assumptions. However, even with perfect L2 assumptions, the obtained IPC is more than 80% lower than the ideal IPC on average. This performance gap must arise from other constraints such a imperfect control flow prediction, constraints on issue width, and finite instruction window sizes.

6.2.3 Effect of Other Constraints

The third bar for every benchmark in Figure 6.5 depicts the obtained IPC when all the processors constraints, except for the issue width and the window size are made ideal. We assume perfect control flow prediction, zero-cycle communication latencies between any two tiles in the processor, and perfect primary caches with infinite bandwidth. Averaged across the suite, the benchmarks exhibit a three-fold increase in performance. In the subsequent section, we explore these constraints in greater detail and provide the relative importance of each.

The difference in performance between the last two bars in Figure 6.5 result from the finite issue width and instruction window size constraints. Issue width constraints impede a machine's ability to exploit available ILP, and high issue widths are necessary to fully exploit the available parallelism in many workloads. Similarly, the size of the instruction window determines the number of instructions from which a machine can discover and exploit parallelism. The results from Figure 6.5 indicate that the restricted issue width and instruction window size together reduce IPC to as low as 15% of the ideal in *tblook* and 48% on average. However, we note that in the presence of other bottlenecks, these constraints may not be as significant

for performance. For example, a larger window size is only useful if the control flow predictor is highly accurate. Similarly, a high issue width is only useful if the instruction window does not stall.

6.2.4 Discussion

In this section, we evaluated the extent of available parallelism in various workloads. We observed that benchmarks exhibited copious parallelism, of which only 16% is exploited by the baseline TRIPS processor. We observed that a perfect L2 cache improves the performance of the baseline TRIPS processor 18% on the average. When other processor constraints such the control flow speculation and operand communication are idealized, we observed that the performance improves by nearly three-fold. Some of these constraints such as control flow mis-speculations and instruction cache misses are common to all processors, whereas others such as the communication latency of a distributed substrate are specific to the TRIPS microarchitecture. A detailed evaluation of these constraints is the subject of the discussion in the next section.

6.3 Where Do Execution Cycles Go?

A number of architectural and microarchitectural constraints inhibit the parallelism achieved by the TRIPS processor. These constraints include branch mispredictions, store/load dependence violations, and structural resource constraints. In Chapter 5, we described how these constraints cause dependences and delays during the execution. This section quantitatively explores the degree to which they affect overall performance. We use critical path analysis to measure the cycles spent by the processor in resolving various dependence constraints. We then identify the constraints that are intrinsic to the design and those that can be mitigated with suitable modifications to the architecture and microarchitecture. The following subsections present

Category	Event
Instruction supply	instruction cache miss, branch misprediction, store/load dependence violation
Data supply	register file access, data cache misses, load deferrals due to conservative store/load ordering
ALU execution	functional unit contention and latency
Operand communication	network hops and network contention
Result commit	register commit and store latency
Distributed control protocols	instruction distribution, completion detection, block commit

Table 6.6: Categorization of various microarchitecture events into different critical path components.

our observations and conclusions.

We organize this section as follows. We first present an overall summary of the major critical path components for various benchmarks. We then present a finer breakdown for each major component on the critical path. Since the contribution to the critical path is not always a true indicator of the effect of certain sub-component on overall performance, we measure its cost using the methodology presented in Chapter 5. We discuss the implications of various results and present some architectural and microarchitectural techniques to alleviate each sub-component's effect. We use *tsim_critical* for all the experiments presented in this section. We configure it with a perfect secondary cache to isolate the bottlenecks in the processor microarchitecture. We report the results for every hand-optimized benchmark and the average results for the compiled SPECint and SPECfp benchmarks.

6.3.1 Critical Path Components

Recall from Chapter 5 that the microarchitectural execution of a program is a sequence of dependence resolutions and the longest dependence path through the program is the critical path of execution. Using *tsim-critical*, we compute the critical path of execution in the baseline TRIPS processor and attribute every cycle on the path to one of six components—instruction supply, data supply, ALU exe-

BENCH	Instruction supply	Data supply	ALU execution	Operand communication	Commit	Block protocols
dct8x8	44.3	6.7	14.2	22.3	3.1	9.3
matrix	7.2	16.3	19.1	44.7	1.6	11.1
sha	8.8	7.3	57.8	24.2	0.4	1.5
vadd	10.5	4.0	27.9	31.4	5.7	20.4
conv	15.0	9.1	16.5	48.7	2.8	7.9
ct	24.8	13.9	12.5	31.1	5.5	12.2
genalg	25.0	5.9	42.1	21.6	0.3	5.1
a2time01	4.5	16.2	40.0	23.9	2.2	13.3
bezier02	7.5	7.6	52.1	21.0	1.2	10.7
basefp01	2.2	2.0	65.9	15.7	3.4	10.8
rspeed01	6.6	9.9	33.9	47.5	0.2	2.0
tblook01	5.6	29.9	27.0	32.8	0.2	4.4
autocor00	5.5	6.5	22.9	53.5	1.6	10.0
SPECint	20.8	26.9	18.2	29.1	0.4	4.7
SPECfp	16.3	33.6	20.5	18.1	2.4	9.2
MEAN	15.6	21.0	26.1	27.3	1.8	8.1

Table 6.7: Major components of critical path.

cution, operand communication, result commit, and distributed control protocols. Table 6.6 describes these components and the specific architectural and microarchitectural constraints that comprise these components.

Table 6.7 presents a summary of the cycles attributed to each component as fractions of the overall execution cycles. We make the following observations from the table. First, instruction and data supply together account for nearly a third of the critical path cycles in the hand-optimized benchmarks and nearly half of the cycles in the SPEC benchmarks on average. These cycles include the overhead of misses in the primary caches, incorrect control and data speculation, and latencies of traversing the respective microarchitecture pipelines. Second, ALU execution accounts for about 30% of the critical path cycles. This component arises from the latencies of executing not only necessary instructions, but also overhead instructions such as fanout and contention for the shared ALU resources. Third, operand communication contributes a significant fraction of the execution cycles, ranging from 15% in *basefp* to as much as 53% in *autocor*. These overheads include the latency

of both sending an operand over multiple hops and the contention encountered at the network routers along the way. Finally, the distributed control protocols present relatively low overheads for execution, ranging from 1–20% and less than 10% on average.

Implications: Except for *commit*, which is shown in the sixth column of Table 6.7 and represents the event of committing architecture state to register files and data cache banks, every component causes non-trivial overheads for execution. It is therefore imperative to craft suitable mechanisms to reduce the overheads of each component. The overheads of instruction supply and data supply are common to all architectures, not just the TRIPS architecture. Techniques such as instruction and data prefetching which have been shown to improve cache performance in other architectures can be adapted for the TRIPS architecture. The overheads of operand communication and distributed block protocols, however, are unavoidable for distributed architectures. As our results show, the distributed block protocols, despite incurring multi-cycle overheads for distributing instructions to the execution units, committing the execution of a block, and flushing incorrect speculation, can be largely overlapped with useful execution in other blocks. However, distributed execution necessitates operand transport between various units and the latencies of such a transport network can inhibit performance significantly. It is necessary to not only minimize the instances of operand transport, but also minimize the latency, whenever it is inevitable.

We now examine each critical path component in depth and discuss various alternatives that can mitigate their effect on performance.

6.3.2 Instruction Supply

Table 6.8 presents the critical path breakdown for the various events that comprise instruction supply in the TRIPS processor. These events include L1 instruction

BENCH	Instruction cache miss	Branch misprediction	Load dependence violation	Fetch pipeline	Total
dct8x8	3.8	0.2	0.1	40.2	44.3
matrix	1.3	0.2	0.0	5.8	7.3
sha	4.0	0.0	0.1	4.8	8.9
vadd	0.9	0.1	0.0	9.6	10.6
conv	8.0	0.1	0.2	6.7	15
ct	12.5	0.2	0.4	11.8	24.9
genalg	14.6	0.3	0.3	9.9	25.1
a2time01	0.8	0.0	0.2	3.4	4.4
bezier02	1.2	0.1	0.1	6	7.4
basefp01	0.6	0.0	0.0	1.5	2.1
rspeed01	2.0	0.1	0.0	4.4	6.5
tblook01	0.8	0.2	0.2	4.4	5.6
autocor00	0.9	0.1	0.0	4.4	5.4
SPECint	8.5	0.3	0.6	11.3	20.7
SPECfp	5.9	0.1	0.7	9.5	16.2
MEAN	5.6	0.1	0.4	9.5	15.6

Table 6.8: Components of instruction supply on the critical path as a percentage of program execution time.

cache misses, branch mispredictions, store/load dependence violations, and the latency to initiate a block fetch at the global control tile. Note that these events merely constitute the constraints for initiating a useful fetch operation in the processor. The actual event of distributing the instructions from the instruction cache banks to the execution units is attributed to the block control protocols. Table 6.8 shows the contribution of each event as a percentage of the overall program execution time.

These results show that in at least one benchmark—*dct8x8*—a large proportion of the instruction supply cycles is consumed in initiating the block fetch at the GT. A portion of these cycles arise from the fetch bandwidth limitation in the TRIPS processor. Recall from Chapter 4 that a new block fetch can be initiated only every eight cycles. This bandwidth corresponds to a fetch of one 32-bit instruction for each of the 16 execution units every cycle. Consequently, even if a successor block is otherwise ready to be fetched and executed, its fetch must be stalled until

eight cycles have elapsed since the fetch of the previous block. Other fetch pipeline cycles are due to dynamic stalls resulting from interlocks between the fetch and the prediction pipeline in the GT. The stall cycles from both of these components inhibit the fetch and execution of future blocks and reduce overall performance.

Primary instruction cache misses also contribute to the critical instruction supply latencies significantly, especially in benchmarks such as *conv*, *ct*, and *genalg*, and the SPEC workloads. Benchmarks *ct* and *genalg* are short and incur most of the misses in the initial warmup phases. The effect of these misses will reduce if the benchmarks run longer. Other benchmarks incur capacity misses in the cache. Branch mispredictions do not appear prominently appear on the critical path. However, as described in Chapter 5, a correct branch prediction has the effect of breaking dependence chains. A broken dependence alters the critical path significantly, often reducing a previously long path to several smaller ones, resulting in marked improvements in performance. Consequently, the true effect of branch mispredictions will be higher than the results depicted in this table. In Appendix C, we present the raw branch misprediction and I-cache miss rates for the various benchmarks.

Load dependence violations are also not critical as the benchmarks rarely mispredict memory dependences. In addition, the first misprediction of a store-load dependence often forces subsequent instances of the same load instruction to execute conservatively, mitigating further mispredictions. We revisit this issue in Section 6.3.3 and show that the conservative execution delays loads unnecessarily and affects performance significantly.

To measure the true effect of various constraints on performance, we performed a cost analysis using the methodology described in Chapter 5. Figure 6.6 presents the results of the analysis. Every benchmark contains two bars representing the percentage speedup obtained if the microarchitecture were able to cache all instructions perfectly, or speculate control dependences perfectly, but not both.

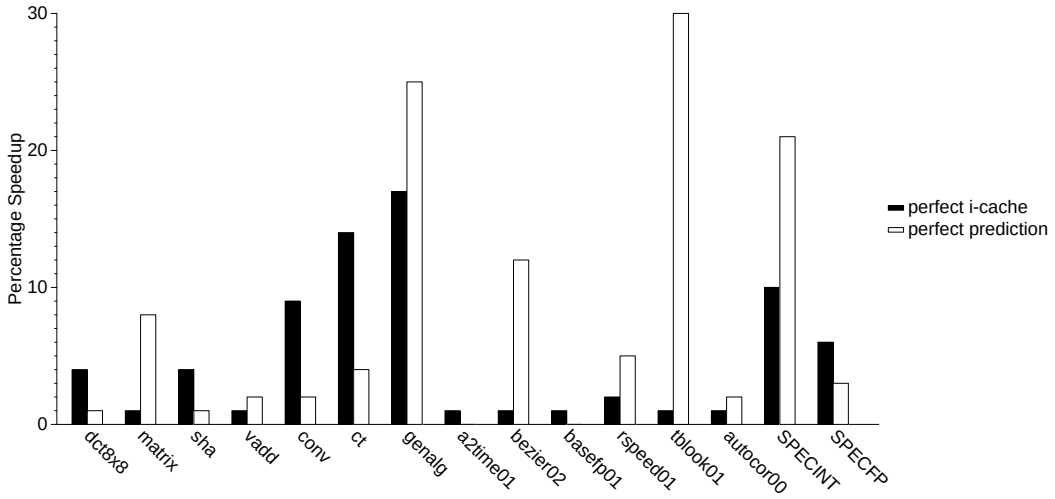


Figure 6.6: Speedup from a perfect front end.

The results are normalized to the execution in the baseline TRIPS processor. They show that an optimized front-end can improve performance significantly in several workloads. Perfect caching improves performance by 6%, whereas perfect prediction improves performance by 9%, on average. The SPEC workloads exhibit greater speedups with perfect caching as they have larger working sets than the hand-optimized benchmarks. Improvements in branch prediction also show good speedups, especially in the SPECint workloads, and in kernels with poor prediction accuracy—*genalg* and *tblook01*.

Implications: SPECint workloads and a few hand-optimized kernels such as *ct*, *dct8x8*, and *genalg* spend more than 25% of the critical path cycles in the instruction supply. Some of the latencies arise from the fetch bandwidth limitations, whereas others are caused by instruction cache misses and poor branch prediction. The front-end is therefore a critical design element in the overall processor microarchitecture and must be enhanced to improve performance.

Enhancing the Instruction Front-end

Improving I-cache performance: To produce fixed-size blocks, the compiler pads each under-full block with NOP instructions. Unless compressed, NOPs reduce utilization at every level of the memory hierarchy resulting in wasteful fills and refills. The variable-sized blocks of the TRIPS prototype architecture offer a rudimentary technique for the compression of small blocks. Using this feature, blocks stay compressed in the secondary cache, but are fully expanded at higher levels. This technique, however, does not improve the primary cache performance. VLIW architectures have faced similar issues and several NOP compression techniques have been examined in prior research [36]. The proposed techniques present the tradeoff of improved cache occupancy, which reduces the number of misses and improves performance, or stretching the fetch pipeline, which degrades performance. Further research is required to explore the suitability of such techniques in the TRIPS architecture.

Alternately, the I-cache performance can be improved by using prefetching techniques. The prediction and fetch pipeline can be fully decoupled using an organization similar to the fetch target buffer [128]. Such an organization will enable the next-block predictor to run ahead of the fetch pipeline and set up a queue of block addresses to fetch. A separate prefetch engine can use these addresses to fill any missing instruction cache lines just in time for the fetch pipeline. However, the efficacy of the prefetch engine will depend on the quality of the branch prediction. The TRIPS prototype processor did not implement this mechanism due to the complexity of managing the prefetch queue and additional support required in the next-block predictor. Further research is necessary into the design of low-complexity prefetch techniques.

Improving fetch bandwidth: The fixed eight-cycle latency between consecutive blocks can be reduced by increasing the fetch bandwidth. This approach, however, requires multi-ported structures in many tiles, which will either stretch cycle time or force deeper pipelining, both of which may adversely affect overall performance. Alternately, a technique called *instruction re-vitalization* can be used to reduce the inter-block fetch stalls [133]. With instruction re-vitalization, each ET could fill its needed instructions at once, by using its instruction buffers as a L0 cache instead of waiting for the ITs to fill them over eight consecutive cycles. The GT can detect if the same static block must be executed immediately again. If so, it can send a single cycle command to all the tiles and instruct them to replenish the instructions from the previous instance of the same block. Such a mechanism would effectively enable the fetch of a new block in successive cycles, if the same set of static blocks (up to eight) execute in a tight loop. Our recent experiments exploring the temporal locality of dynamically executed blocks indicate significant reuse among the recently executed blocks.

6.3.3 Data Supply

Data supply consists of two components—supply of data values from the register file and supply of data values from the memory. Figure 6.7 shows the relative contributions of each component to the program critical path. Register supply includes the microarchitecture latencies to forward values dynamically from producer blocks to consumer blocks, and reading the register file. However, it does not include any operand routing latencies to and from the RTs. Memory supply includes the latency of loads that miss in the primary cache and the latencies incurred by *deferred* loads, which must wait for all prior stores to complete execution before the data tile can satisfy their request. Figure 6.7 shows that both register supply and memory supply contribute in equal measure to the critical path delay in the hand-optimized

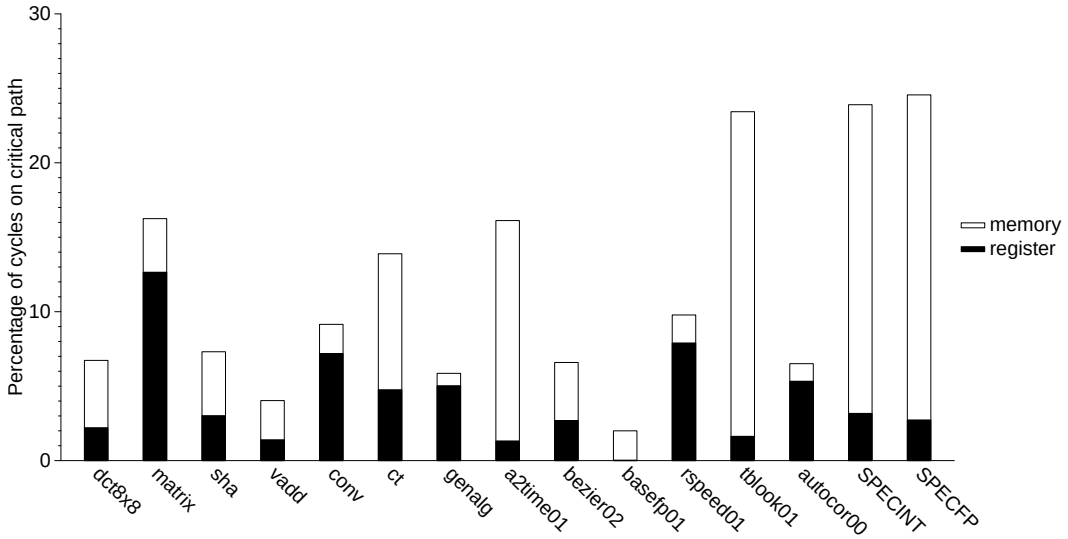


Figure 6.7: Components of data supply on the critical path.

benchmarks. The SPEC workloads, however, are dominated by memory supply.

The register supply latency is dominated by the latency to forward values from producer blocks to consumer blocks. A register value from a producer can be consumed potentially by up to two consumers in each of the successor blocks. Therefore a given register value may be forwarded to potentially up to 14 consumers—two consumers in seven successor blocks. Each RT can satisfy only one consumer every cycle. To forward a register, an RT must first select the register among several candidate registers and then select a consumer. The bandwidth constraint at the RT and the latency of selecting a critical consumer contribute to much of the latency for register supply.

Figure 6.8 presents the constituents of memory supply. Of the three hand-optimized benchmarks where memory supply is critical for performance, *ct* is limited by cache misses, whereas *a2time* and *tblook* are limited by the latencies for deferred loads. SPECint workloads are mostly limited by cache misses, whereas SPECfp workloads are limited by deferred loads. Each DT in the TRIPS processor includes

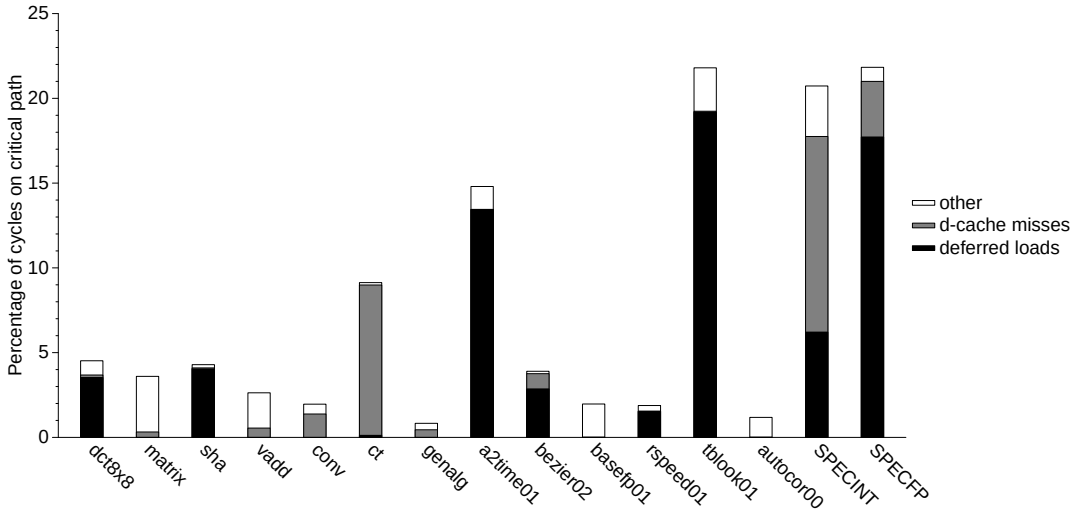


Figure 6.8: Components of data supply from memory on the critical path.

a simple 1-bit dependence predictor [105] that dynamically predicts the dependence between a load instruction and a prior, unresolved store instruction [140]. It defers a load instruction if the predictor indicates that its address will alias with an earlier store instruction. If the prediction is correct, the execution proceeds and exhibits improved performance; otherwise it degrades performance without affecting functional execution. However, if the dependence predictor incorrectly predicts that a load address does not conflict with a preceding store, the execution is functionally incorrect and will result in a dependence violation. The mis-speculation recovery caused by dependence violations and the latencies of deferred loads both affect performance. However, in our experiments, the 1-bit dependence predictor generally caused more deferrals than violations and degraded performance worse than violations.

Implications: For large working sets, memory supply will dominate the latency of all data supply. This fact is evident from the SPEC workloads, where memory supply constitutes more than 20% of the overall critical path in the SPEC workloads.

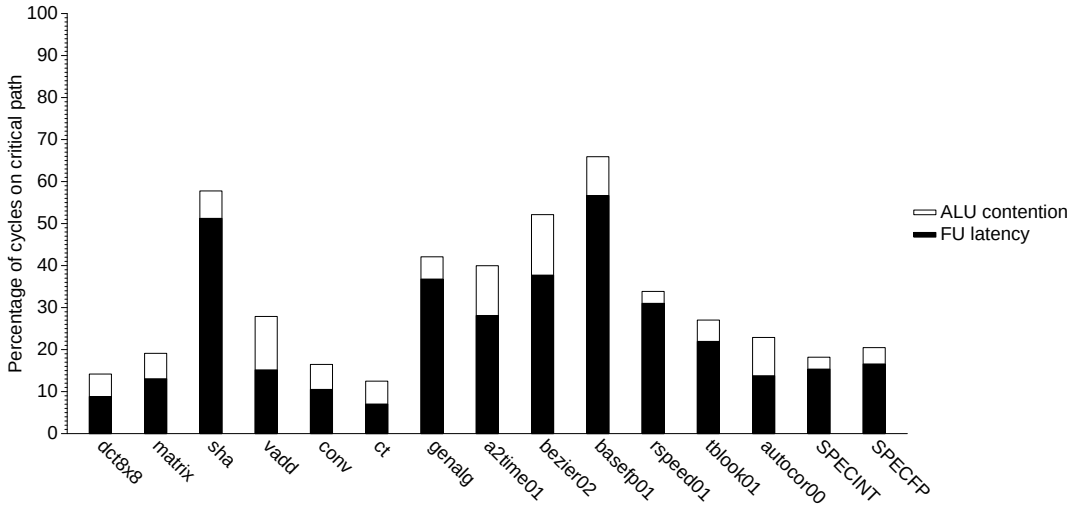


Figure 6.9: Components of ALU execution on the critical path.

As our results indicate, both data cache performance and dependence prediction are important for improving overall performance. Prefetching, which was not implemented in the TRIPS processor, will improve the data cache performance. A better dependence predictor, such as a 2-bit hysteresis predictor, or the store sets predictor [34] will likely reduce the number of deferred loads and improve dependence prediction.

6.3.4 ALU Execution

In the absence of any data value speculation, the execution of a program will be limited only by the longest data dependence chain and the ALU latencies for the instructions on that chain. The ALU execution latency in the TRIPS processor consists of two components—*Functional Unit (FU) latency* and *contention*. The FU latency denotes the latency of an operation at a functional unit. For example, in the TRIPS processor an integer add operation consumes exactly one cycle, where as an integer divide operation consumes 24 cycles. Contention corresponds to the number

of cycles an instruction must stall before issue to ensure that the functional unit is available for executing that instruction. Figure 6.9 presents the effect of these two components on the critical path of execution.

Implications: Ideally FU latencies would constitute the bulk of the program critical path cycles. The results from Figure 6.9 illustrate that 25% of the overall execution cycles are spent in FU latencies. Any reduction must stem from reducing the number of instructions on the critical path. A smaller portion of the execution (7%) is spent in stalls due to ALU contention. Since the hardware does not dynamically re-assign instructions to ETs at runtime, an overload of instructions at one ET while another is free reduces the net execution rate of the processor and increases the program critical path. Figure 6.10 presents the percentage speedup in performance that can result from completely eliminating runtime ALU contention. These results, which show an average 6% improvement, demonstrate the utility of optimizing for ALU contention.

Improving ALU Execution

The FU latencies on the critical path can be minimized by reducing the number of instructions required to perform an operation or reducing the heights of computation trees. The TRIPS compiler, in fact, includes these optimizations: unnecessary sign extension elimination, efficient memory address generation using offsets, and forming balanced computation trees for associative operations. Operand fanout also presents opportunities for reducing the number of instructions on the critical path. These opportunities are discussed further in Section 6.3.7.

The TRIPS compiler must balance the opposing needs of reducing the latency of communication between instructions at different ETs and minimizing the contention at each ET. It does so by modeling the contention for an ET among instructions of not only the same block, but also across multiple blocks. Occasionally,

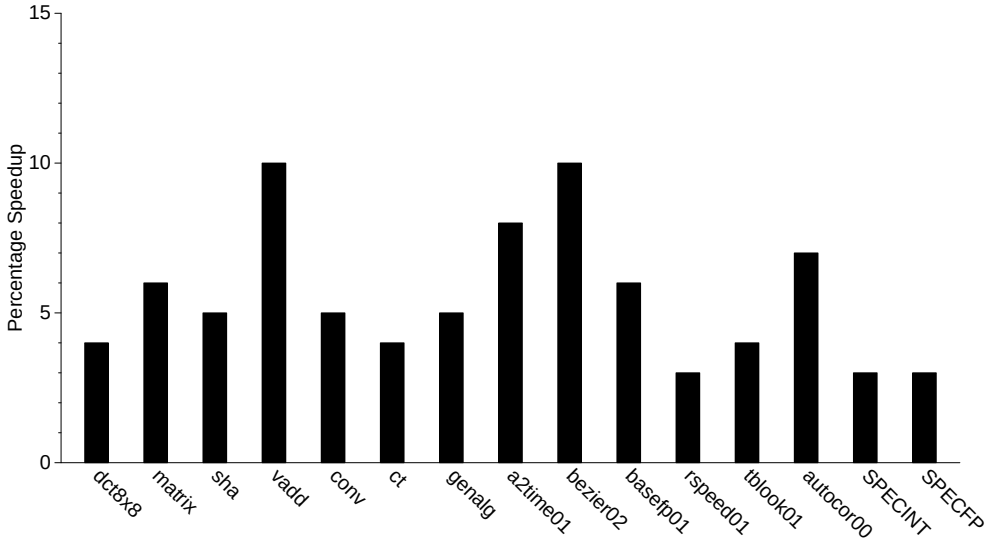


Figure 6.10: Performance effect of ALU contention.

it also prioritizes contention over latency [38]. Various experiments with the scheduler shows that the contention heuristics improve performance significantly over a baseline scheduler that does not consider any contention [38, 108]. Despite using these heuristics in the scheduler, runtime contention is inevitable. Any further improvements must come from increasing the execution bandwidth in each ET. The performance speedups depicted in Figure 6.10 range from 1%–10%, indicating that the scheduler is effective in reducing contention in most cases.

6.3.5 Operand Communication

Section 6.3.1 presented operand communication as a critical bottleneck for performance. In this section, we examine the constituents of operand communication and the latencies that affect performance. Recall that the operand network transports different types of data in the TRIPS processor. It transports register values from RTs to ETs, temporary results values between producer and consumer instructions in different ETs, load addresses and data between ETs and DTs, and finally output

BENCH	Temporary	Register inputs	Register outputs	Load	Branch	Store
dct8x8	12.3	1.0	1.8	2.4	1.2	3.7
matrix	24.3	1.1	3.0	14.4	0.7	1.2
sha	19.2	0.2	4.0	0.7	0.1	0.0
vadd	19.5	1.0	0.8	7.9	0.5	1.7
conv	9.3	16.9	16.0	3.5	0.5	2.5
ct	16.9	1.8	2.1	2.7	1.1	6.5
genalg	12.3	2.0	3.7	1.6	1.8	0.0
a2time01	17.7	0.3	1.4	3.1	0.0	1.4
bezier02	14.7	1.3	3.3	0.8	0.6	0.3
basefp01	9.9	0.0	0.0	3.4	0.0	2.4
rspeed01	22.3	11.9	12.2	0.5	0.5	0.1
tblook01	20.3	0.9	2.5	7.0	1.9	0.3
autocor00	31.3	0.4	5.2	16.0	0.6	0.0
SPECint	13.0	0.8	4.2	9.3	1.4	0.5
SPECfp	7.4	1.5	2.4	4.1	0.3	2.3
MEAN	13.6	2.1	3.7	5.7	0.8	1.6

Table 6.9: Types of operand communication. Numbers represent the contributions of each type of operand communication to the overall critical path of the program.

values—registers, stores, and branches—from ETs to their respective destinations. Table 6.9 provides a breakdown of these types of communication that occur during execution.

Not surprisingly, the principal components of operand communication are temporary result values, load addresses and data, and register outputs. In general, temporaries contribute less to the overall communication in the SPEC workloads. This result is not surprising, since the compiled SPEC benchmarks typically have fewer instructions in each block, resulting in less computation and communication among instructions of the same block. Register outputs are critical because typically outputs from one block are inputs to another, thus forcing the register output communication on the overall critical path of the program. Branch communication is rarely critical, unless the corresponding branch resolves a misprediction. Communication of stores is also not critical because stores rarely forward their data to other loads within the window of execution [141]. However, if the window becomes full, deallocation of the oldest block becomes critical, which places the store commu-

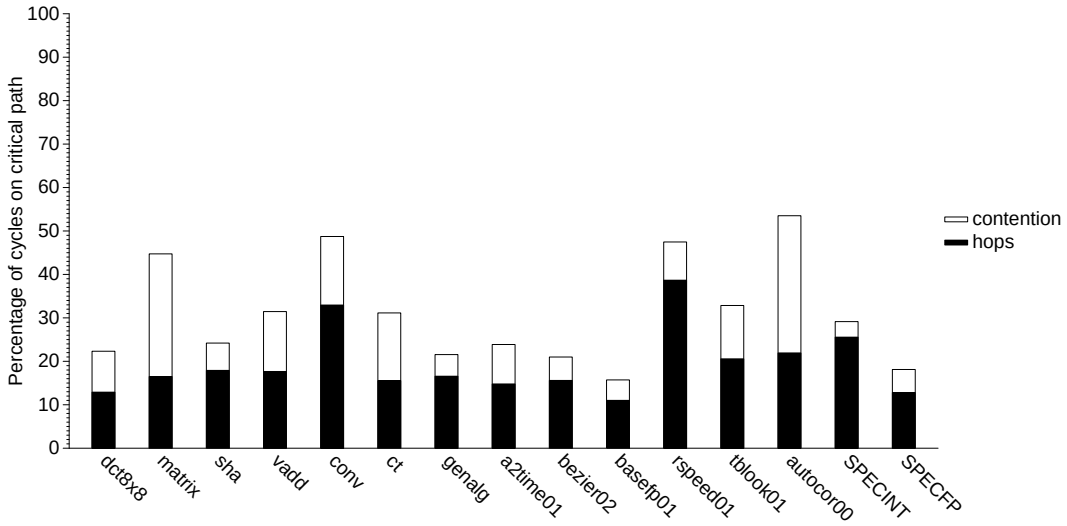


Figure 6.11: Components of operand communication latencies on the critical path.

nication and the subsequent completion and commit protocols for the oldest block on the critical path.

There are two components to the communication latency on the operand network. The first component is the number of hops between the producer and a consumer. Since each hop takes at least one cycle, the proximity of a producer to its consumer directly affects the latency of communication. The second component is the contention in the network. When multiple communication packets simultaneously attempt to use the same network link, the router controlling that link must arbitrate and select one packet, which increases the latency for other packets. Contention also stems from the port constraints at the input and output interfaces of the producer and consumer tiles on the operand network. For example, if an ET cannot accept more than one operand from the network each cycle, the router must arbitrate among multiple incoming packets for delivery to the ET. Figure 6.11 provides the contribution of both the hop counts and contention to the overall critical path.

Implications: Communication latencies are a downside for distributed architectures and unless mitigated, can affect performance significantly. From Figure 6.11, we observe that in the absence of contention the communication latency is strictly determined by the number of hops between the producer and the consumer and accounts for nearly 20% of the cycles on the critical path on average. Contention is also critical for performance as evident in a few benchmarks such as *matrix* and *autocor*. Reducing both components of operand communication is essential for improving the performance of distributed execution.

Improving Operand Communication

The number of hops for communication can be reduced by keeping producers and consumers of a communicating pair in close proximity. This optimization is the primary objective of the TRIPS instruction scheduler. Similar to ALU contention, the scheduler also models network link contention to reduce dynamic link contention. But unless the entire critical dependence path is scheduled on the same ET, communication latencies are inevitable. Even so, instructions need to communicate with RTs and DTs to read and write architectural state. Unless these tiles are integrated into the ET, communication latencies cannot be hidden from the critical path.

Further improvements in operand communication may have to come from alternative network topologies and routing protocols. A higher connectivity network can reduce the number of hops. A higher bandwidth network or a network with an adaptive routing protocol can reduce contention for the network links. The number of input and output ports on the interfaces with network must also be increased at each tile to reduce the contention for incoming or outgoing packets. Figure 6.12 presents the potential speedup that can be obtained if the different components of operand communication can be completely hidden by other execution. While not realistically attainable in practice, these results present an upper bound for all opti-

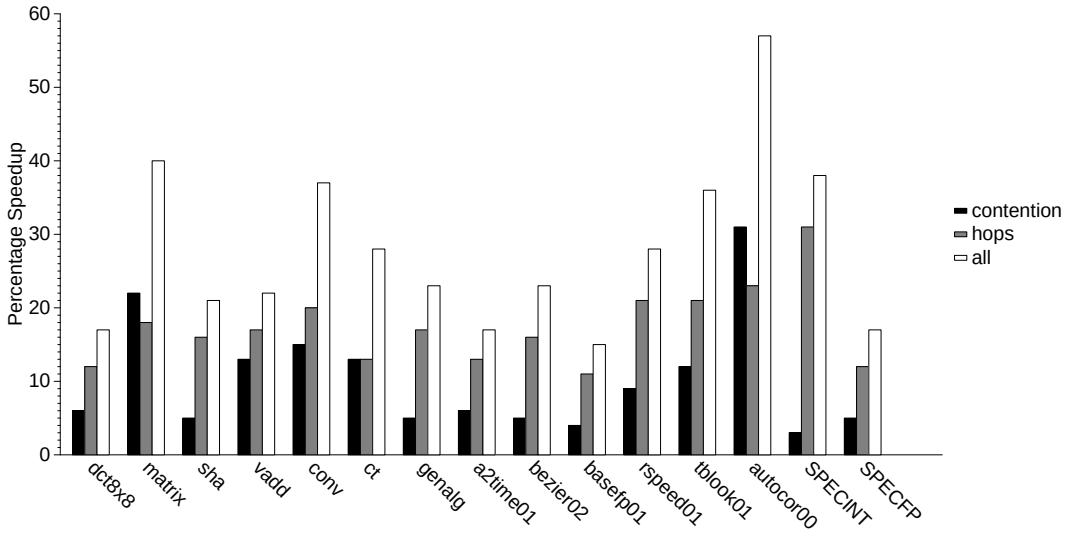


Figure 6.12: Speedup from perfect operand communication.

mizations that address operand communication. Most of the performance speedups arise from reducing the number of hops. SPECint benchmarks show the most improvement from fewer hops compared to contention, as the blocks are relatively small, causing fewer performance losses from contention.

6.3.6 Distributed Protocols

Section 6.3.1 presented the contribution of the distributed control protocols to the critical path of execution. In this section, we provide a finer breakdown of the block control protocols and discuss their effect on overall performance. Figure 6.13 provides the relative contribution of three block control protocols to the critical path. The bottom bar for each benchmark depicts the proportion of overall cycles spent in distributing instructions from the ITs to the ETs. The middle and top bars represent the cycles spent in detecting when a block has completed execution and when a block has committed its execution results respectively. The actual latency of committing architectural state within each RT or DT is measured by the column

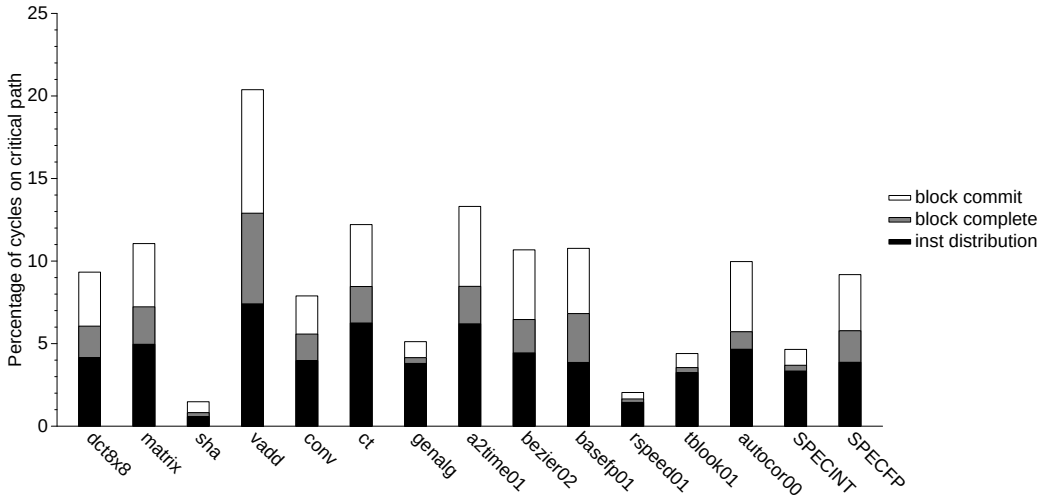


Figure 6.13: Components of block control protocols on the critical path.

titled *Commit* in Table 6.7. Figure 6.13 measures just the protocol latency of sending control signals back and forth with the GT.

Implications: Taken together the distributed control protocols account for less than 10% of the overall execution cycles, indicating that these protocols can be overlapped with useful execution frequently. Of the three protocols, instruction distribution is typically the most critical and accounts for nearly 5% of the cycles. Block commit protocol is generally more critical than block completion. This result is because commit involves both a request and an acknowledgement that together require at least eight cycles for completion. On the other hand, block completion notification can happen even in a single cycle, if the closest RT or the DT receives the last of the block outputs and notifies the GT that the block has completed execution.

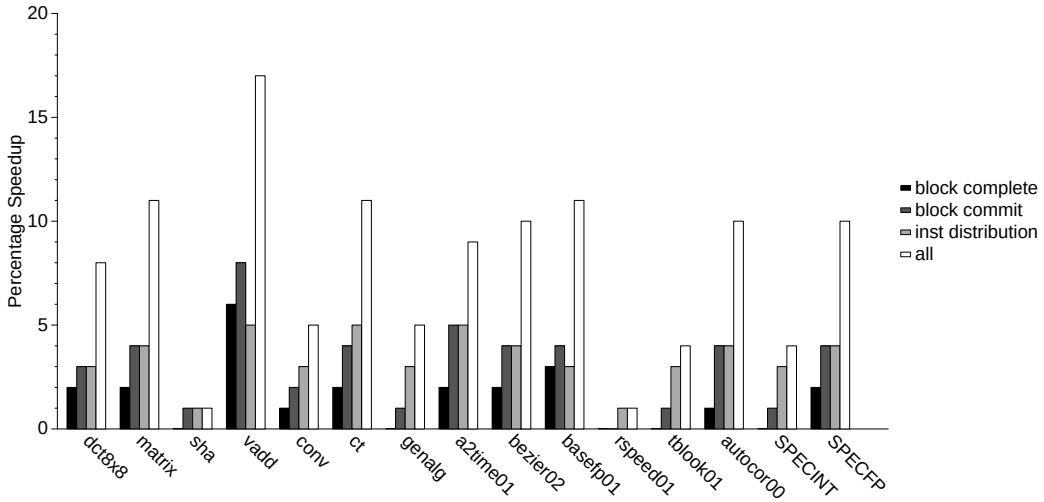


Figure 6.14: Speedup from perfect distributed protocols.

Improving block control protocols

Mis-speculations and instruction window stalls typically expose the block control protocols on the critical path. For example, the correct fetch following a mis-speculation recovery exposes instruction distribution on the critical path. Similarly instruction window stalls expose distributed completion and commit for the oldest block and instruction distribution for its replacement block on the critical path. Any technique that reduces mis-speculations and instruction window stalls reduces the effect of the block control protocols.

Figure 6.14 provides the effect of each protocol on overall performance. If the latencies of all protocols were completely overlapped with useful execution, performance would improve by 8% on the average. Most of these improvements result from the perfect overlap of instruction distribution and block commit protocols with useful execution. Data intensive benchmarks such as *vadd* and *ct* exhibit greater speedups, as the limited store bandwidth in the microarchitecture frequently causes frequent instruction window stalls. Therefore hiding the latency of the distributed

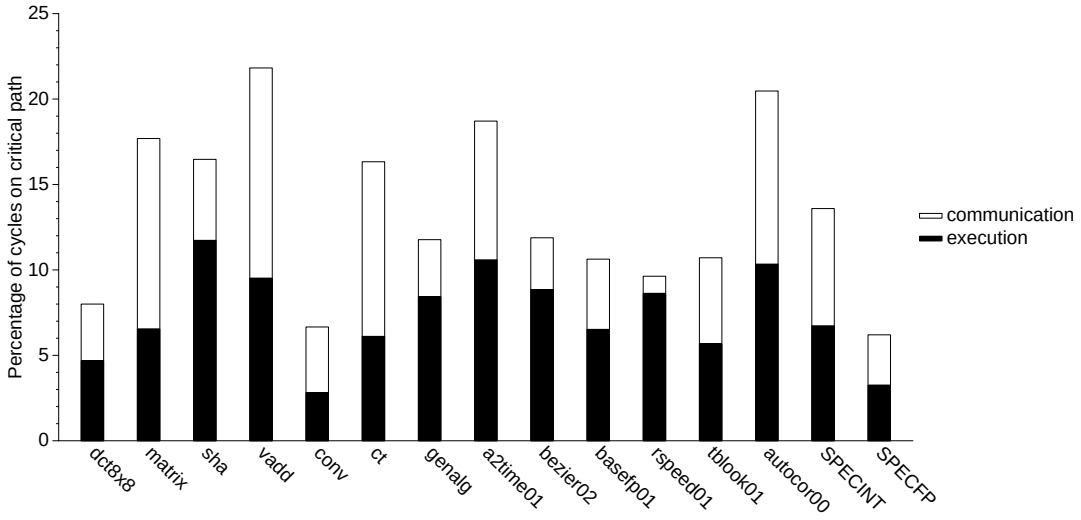


Figure 6.15: Components of operand fanout on the critical path.

commit and fetch protocols improves performance in these benchmarks significantly.

6.3.7 Operand Fanout

Section 6.3.5 and Section 6.3.4 described the combined effect of all ALU operations and all data communication on performance. In this section, we separate operand fanout from other components on the critical path and present its effect on performance. The distribution of a single result value to many consumers requires a software tree of move instructions. These move instructions utilize ALU resources during execution and network resources during communication. Figure 6.15 presents the contribution of these two components to the critical path of execution. The top portion represents the critical path latency of communicating data operands to and from the fanout tree, while the bottom portion represents the critical path latency of executing the fanout tree. In general, the execution of fanout instructions has a greater effect on performance than the communication of values through the fanout tree.

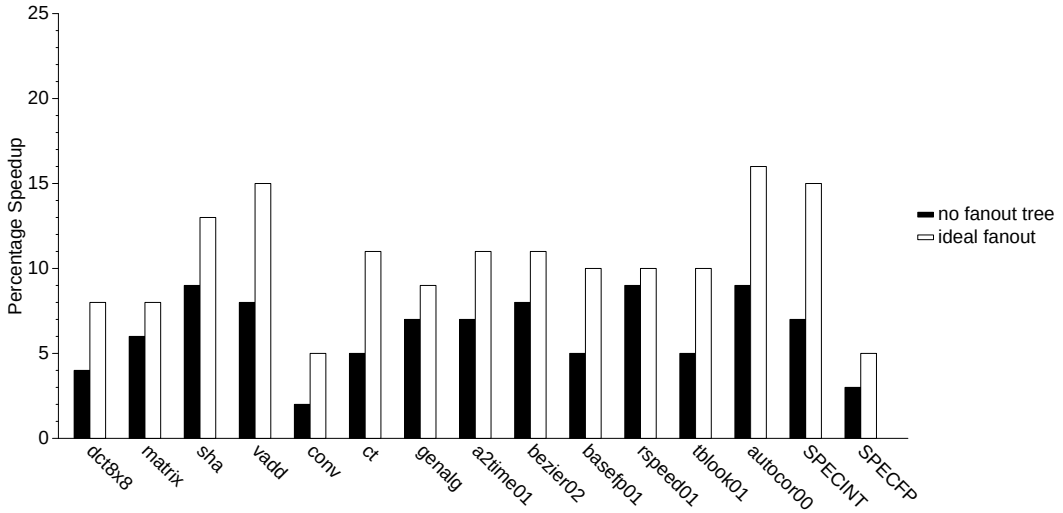


Figure 6.16: Speedup from a perfect fanout.

Execution of fanout instructions can be completely eliminated if all consumers can be represented with the same producer instruction. Communication of fanout values can be eliminated if the following conditions are satisfied: a) all consumers encoded with the producer instruction, b) all consumers scheduled at the same node as the producer, and c) the ET supports a full bypass of the result value from the producer to all consumers. While these assumptions are optimistic, they are useful for computing an expectation for performance improvements with various fanout optimizations. Figure 6.16 presents the percentage speedup that can be obtained if the fanout operations were completely off the critical path. The bar labeled *no fanout tree* represents the speedup obtained if no fanout instructions were executed. The bar labeled *ideal fanout* represents the speedup obtained, if in addition to the execution of fanout instructions, communication overheads of fanout values were removed. As shown in the figure, eliminating all fanout overheads improves performance by as much as 16% in *autocor*, 15% in SPECint, and about 11% on the average. Eliminating just the execution of the fanout instructions, on the other

hand, improves performance by 6% on the average.

Implications: Fanout is an essential overhead of dataflow ISAs. In a few SPECint workloads, we observed that eliminating fanout can improve performance by more than 20%. Fanout also has an indirect effect on performance. Fanout instructions occupy resources within a single block preventing more useful instructions from filling the block. Therefore, they cause an increase in the number of static blocks, which increases the instruction cache pressure. Therefore, the net effect of fanout is likely to be closer to or greater than the results depicted in Figure 6.16.

Optimizing Fanout

Fanout in the TRIPS architecture can be reduced in the following ways:

ISA optimizations: Certain types of operands typically require a greater fanout than others. For example, predicates, base addresses for loads and stores, and constants such as 0 or 1, typically need fanout to several consumers. Selective optimization of these cases can reduce the number of fanout instructions. One such optimization is the use of new predicate-defining instructions that accept a special data operand to specify additional targets. For example, consider a new instruction: `teqz pred, val, target`. This instruction can implicitly compare an input operand with zero and send the resulting predicate to several consumers specified in the target operand. Using the target encoding format specified in Figure 4.1, a single 64-bit operand can specify up to seven additional consumers. Combined with the target specifiers within the instruction, this approach has the potential to significantly reduce the height of the fanout tree.

The target encoding format can also be revised to specify more consumers. Most targets in the prototype TRIPS ISA are specified with their complete coordinates in the execution window. Instead, the target specifier could combine location

information of the producer to compute the coordinates of the target instruction implicitly. For example, instead of using two 9-bit specifiers to represent two targets, the ISA can use a 16-bit mask to specify which of the equivalent operand slots in other tiles are consumers of the same result. This approach reduces the size of the fanout tree. In fact, mid-way during the development of the TRIPS prototype we added two special instructions, `mov3` and `mov4`, which implicitly specify three and four targets respectively. Figure 4.1 presented the encoding formats for these two instructions, which enforce certain restrictions on the target locations, but specify additional targets. The bit-mask target encoding is a logical extension of this approach.

Network optimizations: Efficient encoding of targets reduces the number of instructions required for fanout. However, it does not reduce the number of dynamic network packets required for distributing a value from a single producer to all consumers. The network essentially performs a multicast operation as multiple point-to-point communication operations. This approach not only stalls a producer tile until the multicast is complete, but also increases contention in the network. An alternative approach can implement multicast operations efficiently in the network itself. The network router at the producer can accept a value to be multicast and a specifier that encodes the consumers of the value. Subsequently, the network can dynamically replicate packets as needed and send the value to all the consumers. This approaches reduces stalls at the producer and the total number of packets in the network, both of which reduce contention and improve performance. However, this benefit must be balanced against the increased complexity of the router.

Instruction optimizations: Peephole optimizations can reduce the number of instructions needed for fanout. In Chapter 5, we described two techniques to reduce the fanout of data and predicate operands—predicate combining and ϕ-merging.

These techniques combine common operands to produce a single fanout tree instead of separate fanout trees for each operand. A reduction in fanout not only reduces the overhead of dataflow encoding, but also improves performance by allowing the formation of larger TRIPS blocks.

6.3.8 Discussion

The results from this section show that several microarchitectural constraints play significant roles in determining the performance of a program. Like most architectures, improved instruction supply and data supply in the form of better branch prediction and caches can improve performance. In addition, future improvements in the TRIPS architecture must come from reducing the overheads of distributed execution, specifically operand communication, and dataflow execution, specifically fanout of data values.

6.4 Summary

We began this chapter by establishing the performance of the TRIPS architecture in relation to a conventional superscalar architecture. We presented the results from the TRIPS hardware and examined its ability to exploit a wide execution bandwidth and a large instruction window. We observed that on a set of hand-optimized benchmarks, the TRIPS hardware sustains average speedups of 3× over the Alpha 21264 architecture. However, on the compiled SPEC benchmarks the TRIPS architecture exhibits slowdowns in most benchmarks and improves performance only in three of the benchmarks—*mcf*, *mgrid*, and *swim*.

We then explored the potential for exploiting parallelism in various workloads and compared it with the instruction throughputs obtained in practice. We observed that the TRIPS processor attains only 16% of the available parallelism on average. We observed that issue width and instruction window constraints together reduce

achievable parallelism by nearly 50%. Of the remaining constraints, we observed that instruction supply, data supply, operand communication, and operand fanout all play significant roles in determining the performance of a program. The latency of distributed execution degrades performance significantly. Operand communication in the distributed substrate contributes 15%–53% of the critical path cycles. The block control protocols, however, can be overlapped with block execution in many cases and contribute to less than 10% of the critical execution cycles on average. We also presented a few solutions to alleviate the effect of these constraints and improve overall performance in the TRIPS processor.

Chapter 7

Conclusions

The microprocessor industry is at an interesting inflection point. While the underlying devices continue to become smaller, constraints such as power, reliability, and on-chip wire latency are changing the design parameters of modern microprocessors. Recent years have seen the demise of clock frequency growth primarily due to its taxation on power consumption. This trend has forced architects to focus on concurrency exclusively to meet the ever-growing performance needs of various applications. At the same time, growing wire-delays coupled with increased power and design complexity have limited the scalability of conventional superscalar processors to wider instruction issue, resulting in the virtual abandonment of single-thread concurrency in favor of exploiting multi-thread and data-level concurrency. While this approach pays dividends in the short term, Amdahl's law and the lack of single-thread performance will ultimately inhibit its ability to increase performance. To address this problem, we undertook the challenge of designing a scalable, wide-issue, large-window processor that mitigates complexity, reduces power overheads, and exploits ILP at future wire-delay dominated technologies. This dissertation examines the design and evaluation of such a processor.

7.1 Dissertation Contributions

This dissertation presented the TRIPS architecture for improving single-threaded performance at future wire-delay dominated technologies. It is an Explicit Data Graph Execution (EDGE) architecture, which expresses dependences explicitly in the ISA. The ISA organizes program computation into large groups of instructions called blocks. Each block encodes the dataflow graph of its computation and executes atomically in the hardware. The encoding uses dataflow arcs, instead of register names, which enables the hardware to enforce intra-block data dependences using direct producer-consumer communication. This technique eliminates several per-instruction overheads of register renaming, dynamic scheduling, and complex operand bypass.

The microarchitecture partitions all of the traditionally-centralized structures and connects the partitions with point-to-point networks. It consists of a two-dimensional array of replicated tiles that each implement one of the following functions—ALU execution, register storage, data caching, or instruction caching. The tiles exchange information using multiple interconnection networks, and together implement the functionality of one larger, powerful uniprocessor. Such an organization provides a scalability path to larger processor architectures. The addition of more functional units or cache banks only involve the replication of the desired functionality and an extension of the interconnection networks to additional nodes, without major design rework of the entire microarchitecture.

This dissertation presented the details of a hardware prototype of the TRIPS architecture, and in particular, the author's contribution to its implementation—global control functions and performance validation. The prototype hardware implements all major control functions such as fetch, commit, and flush using a set of simple master-slave protocols. A centralized control logic tracks the execution state on behalf of the entire processor and initiates various control protocols by sending

signals on a set of simple point-to-point networks. The rest of the microarchitecture tiles respond to the signals by performing the desired functionality locally. The slave tiles operate independently of each other, and the protocols avoid any global synchronization. Furthermore, the protocol operations are overlapped with each other and across multiple blocks, which reduces their effect on the overall execution critical path.

The TRIPS prototype chip was implemented in a 130 nm ASIC technology, and it consists of more than 170 million transistors. It contains two TRIPS processors with each capable of executing 16 out-of-order operations from an in-flight window of 1024 instructions. We described the methodology of verifying the performance of the prototype implementation, and in particular, the latency of various critical microarchitecture events. We also described the correlation of the implementation with a high-level performance model and showed how the two models match within 4% on a large set of microbenchmarks. At the time of writing this dissertation in May, 2007, the TRIPS chip had taped out and the first manufactured silicon parts are fully operational, executing several single-threaded workloads and MPI-based multi-threaded workloads, at a peak clock frequency of 366 MHz and a peak power consumption of 45 Watts.

We then described a detailed performance evaluation of the TRIPS architecture. We described our evaluation methodology and in particular, the development of various hand-optimizations to produce high-quality programs. We also described the development of critical path models to identify the fine-grained bottlenecks of distributed execution in the microarchitecture. We described the complexity of evaluating the interactions among a large set of concurrent events and described the algorithms that offer orders of magnitude speedups in evaluation time. We then measured the raw performance of the TRIPS hardware and observed that it provides good speedups over conventional architectures on a set of highly hand-

optimized benchmarks. Finally, we evaluated the potential for high parallelism in the TRIPS architecture, the overheads that inhibit parallelism, and suggested suitable optimizations for enhancing performance.

7.2 Performance of the TRIPS Architecture

The TRIPS processor exploits its wide execution bandwidth and large instruction window to sustain significant parallelism. On a set of hand-optimized benchmarks, the TRIPS hardware sustains speedups in the range 0.9×–4.9×, and 3× on average when compared to the Alpha 21264 microarchitecture. On the same workloads, the TRIPS processor sustains sustains average IPCs of 4.0 and more than 6.0 in a few benchmarks. The large window of instructions, wide dynamic issue, and higher bandwidth to memory contribute to the performance benefits of the TRIPS architecture.

On the compiled workloads the results of the TRIPS architecture are less impressive. In fact, the TRIPS architecture exhibits an average 17% slowdown and as low as 77% slowdown across the SPEC workloads compiled by the TRIPS toolchain. It improves performance significantly in only three benchmarks—*mcf*, *mgrid*, and *swim*—where the speedup exceeds 50%. The poor performance in the compiled workloads is largely due to the increased number of instructions executed by the TRIPS processor. As the TRIPS compiler matures and implements all necessary optimizations, including those described in Chapter 5, we expect the performance of the compiled workloads to improve significantly.

The benchmarks in our evaluation suite exhibit abundant parallelism. Some of them are embarrassingly parallel and limited only by the execution resources available in the machine. In general, issue width and instruction window size restrictions together inhibit parallelism by nearly 50% on the average. Compared to an ideal processor that has a 1024-entry instruction window and 16-wide OOO issue, the

TRIPS processor obtains only a third of the available parallelism. The remainder of the parallelism is lost due to various constraints in the microarchitecture.

We evaluated the effect of various microarchitectural constraints on performance using a detailed critical path analysis. As in other architectures, the instruction front-end performance, including branch prediction accuracy and instruction cache hit rates, affects performance significantly. Data cache performance also affects performance significantly in a few benchmarks. Together, the front-end and data cache performance account for a third of the cycles spent during execution in the hand-optimized benchmarks and nearly half the cycles in the SPEC benchmarks. To a large extent, the distributed control protocols for fetch, completion detection, and commit do not present serious bottlenecks for performance, contributing to 10% of the critical path cycles on average. The large instruction window enables the TRIPS processor to amortize these overheads and overlap useful execution with the latencies for protocol communication. However, operand communication, which is a necessity for distributed execution, significantly affects performance and accounts for a third of the cycles spent during execution on average. Both the number of communication hops and contention in the network contribute to the latencies for operand communication. Finally, fanout of operands due to limited target encoding space in the ISA present non-trivial overheads for performance. The performance losses due to fanout amount to 11% on average across the benchmark suite, and more than 20% in a few SPECint benchmarks.

7.3 Improving the TRIPS architecture

Further performance improvements in the TRIPS architecture must come from alleviating several overheads. As in other architectures, better branch prediction and prefetching can improve the performance of the front-end microarchitecture and the data cache and improve the overall performance. With respect to the instruction

front-end, future implementations should focus on improving the utilization of the cache and also overcoming the fetch bandwidth limitations. Additional improvements should come from solutions for reducing the operand fanout and reducing the latency of operand communication. However, there is no single bottleneck that—if addressed adequately—will improve performance of the TRIPS architecture dramatically. Future performance improvements will require a concerted effort in the architecture, microarchitecture, and the compiler. Based on the results presented in this dissertation and drawing from design experience, we present several revisions for future generations of the architecture and microarchitecture.

The next revision of the TRIPS microarchitecture should address the following:

- De-coupled front-end: The front-end architecture should decouple the prediction and the block fetch pipelines. This approach will enable aggressive instruction prefetching [128] and help tolerate the penalty of I-cache misses.
- Dependence predictor: The current 1-bit dependence predictor, although simple to implement in hardware, conservatively predicts a conflict too often, even if none exists. The dependence predictor should be enhanced in simple ways to include hysteresis or re-designed to implement more advanced techniques such as store-sets [34].
- Fetch bandwidth: Although not a serious bottleneck in many benchmarks, the fetch rate of one block every eight cycles constrains the performance occasionally. In addition, since TRIPS blocks are never 100% full, the effective fetch rate in the TRIPS prototype processor lags behind the execution rate offered by the microarchitecture. Instruction re-vitalization should be considered to enhance the fetch rate of blocks belonging to tight loops.

Subsequent generations of the TRIPS architecture and microarchitecture should

address the following:

- Instruction cache compression: The primary instruction cache should be modified to retain compressed block encodings. Compression improves the cache utilization, but increases the latency of instruction fetch and complicates the overall distributed cache management. Future designs should develop efficient solutions that balance these opposing requirements.

- ISA enhancements: The ISA should include more instruction formats to provide a better encoding of targets. We make two recommendations: a) encoding targets in data operands, and b) new target formats to represent more than four targets. The operand routers should change suitably to accommodate these enhancements. Other ISA enhancements such as specialized immediate-forms and additional support for signed arithmetic can help reduce the number of instructions executed in the TRIPS processor.

- Operand routing: Multicast support should be considered to reduce the fanout overhead in the network. In addition, adaptive routing protocols with deadlock avoidance or deadlock detection should be considered to reduce the contention.

- Contention: Both ALU and operand network contention present some overheads for performance. Although, their effect is not as pronounced as other constraints, reducing contention has the potential to offer more than 20% improvement in performance in a few benchmarks. Future implementations may consider increasing both the operand network bandwidth and the execution bandwidth. However, this decision should be weighed against the increased power and area complexity.

Looking further, the biggest challenge for the TRIPS architecture, and distributed microarchitectures in general, will be operand communication hops. Future orga-

nizations should attempt to create and exploit locality of communication by distributing dependent instruction chains to only a small neighborhood of execution tiles. The microarchitecture should consider migrating other communicating entities closer together. This optimization includes the migration of cache banks and register file banks closer to the execution units. It has the potential to reduce the latency to caches and register files and increase bandwidth, but introduces new challenges. For example, data partitioning not just among the cache banks, but also among the register banks will determine the data supply latencies. Suitable algorithms should be developed to isolate dependence chains and co-locate instructions with the data they access. Pure hardware techniques that dynamically migrate data closer to the accessing instructions may not be viable due to the complexity of maintaining coherence and dynamically locating data among a large set of cache banks. In addition, they necessitate efficient mechanisms to enforce the correct program order for loads and stores [141].

The co-location of data caches and register files with each execution unit, or even among a set of execution units also increases the latency of communication between dependent instructions on different execution units. It results in either physically larger tiles, which increase per-hop latencies, or physically farther tiles, which result in additional hops. Further research is necessary to explore the tradeoff of increased latency among the functional units and reduced latency to the caches and register files.

7.4 Concluding Thoughts

The TRIPS prototype processor is a functional, physical embodiment of an EDGE architecture. Its successful development is a testament to the fact that a microarchitecture in which distributed communicating components cooperate to execute a single thread of application is a feasible approach for exploiting concurrency. This

dissertation has shown that on a handful of aggressively hand-optimized benchmarks the performance of the TRIPS processor is superior when compared to a best-of-breed ILP architecture. It has also demonstrated that except for data communication, which is a necessity for any distributed architecture, distributed execution does not present significant overheads for performance.

Before it can gain acceptance in the mainstream microprocessor world, the TRIPS architecture must overcome a few technical challenges. An open question is whether the performance of compiled code will measure up to hand-optimized code. The availability of functional TRIPS hardware will undoubtedly reduce the development-optimization loop in the compiler. However, history has shown that the maturation of any compiler technology typically occurs only after several years of concerted development. It is quite likely that the TRIPS compiler will follow that path.

A second open question is the power efficiency of the TRIPS architecture. In this dissertation, we argued that the architecture offers several benefits—reduced per-instruction overheads and elimination of dynamic dependence check hardware—for reducing power consumption. However, we have not evaluated these benefits quantitatively. Current hardware measurements indicate a peak power consumption of 45 W at a clock frequency of 366 MHz for the entire TRIPS chip. Most of this power is spent in the clock distribution network and idle dynamic power. On the one hand, higher clock frequencies will certainly increase the power consumption to greater than 45 W. On the other hand, the adoption of a power-aware design methodology, including clock gating and less leaky devices, will reduce the power consumption. The compiler algorithms for predication and hyperblock formation also present tradeoffs for power and performance. A comprehensive evaluation is necessary for understanding the various components of power consumption in the TRIPS processor.

An interesting dimension to this power-performance tradeoff is the adaptability of the architecture to available parallelism in the program. Our results indicated a wide variance in parallelism among different programs. Even within a single program, different execution phases exhibit a wide variance in available parallelism [144]. The hardware can exploit this phenomenon to improve power-efficiency by dynamically adjusting resource utilization depending on the availability or the need for parallelism. For example, instead of using 16 ETs to perform a serial dependence computation, the hardware can utilize just one or two to perform the same computation, thus achieving the same performance at reduced power consumption. In a different phase of the same program, the hardware can switch to utilizing all 16 ETs for improving parallelism. The precise mechanisms and policies for adjusting resource utilization are subjects of future research.

Finally, object-code compatibility will also affect the adoption of TRIPS technologies for general-purpose computing. Programs compiled for one generation of the TRIPS architecture must be supported by subsequent generations. VLIW/EPIC architectures faced similar issues and several software and hardware techniques have been proposed [37,48,60,98,127]. Unlike VLIW, the TRIPS architecture determines only the placement of the instructions and not the execution order, which makes retargeting pre-compiled TRIPS object code for new TRIPS hardware a relatively simpler problem. One simple solution may choose to compile for a generic microarchitecture organization and remap the instruction placements for different organizations dynamically in the hardware. For example, the 3-D coordinates for a slot in the 128-entry instruction window can be re-interpreted to accommodate a 4×4×8 organization, 2×2×32 organization, or a 8×8×2 organization. Alternate organizations may, however, require techniques such as dynamic recompilation to support backward compatibility [48].

Ultimately, improvements in single-thread performance must come from co-

operative solutions at all layers of the application execution stack. Exclusive solutions at any single layer are likely to be unsuccessful. For example, pure programmer-managed concurrency requires unconventional programming models, and complicates the development and maintenance of software, which is already a problem of leviathan proportions. Pure hardware solutions will suffer from inefficiency as they must expend power to discover concurrency patterns in the program. Successful solutions are likely to employ mechanisms at all levels—program annotations, compiler hints, and hardware techniques—without disrupting programmer productivity. This dissertation presented the TRIPS architecture that uses both the compiler and the hardware to enhance the scope of parallelism among a contiguous stream of instructions. Future architectures must look beyond a contiguous window, perhaps using programmer hints, and adapt to changing granularities of concurrency to not only improve performance, but also attain power-efficient execution.

Appendices

Appendix A

Comparing *tsim-proc* and the TRIPS Hardware

In Chapter 4, we reported the results of performance correlation between *tsim-proc* and *proc-rtl*, which is the RTL-level simulator. We reported that after normalizing the L2 memory system, the performance results from *tsim-proc* and *proc-rtl* match within 4% on a large number of microbenchmarks. In this section, we examine the errors in performance estimation in *tsim-proc* when compared to the hardware.

Table A.1 presents our results. Positive differences indicate over-estimation by *tsim-proc*, whereas negative differences indicate under-estimation. In many workloads, *tsim-proc* overestimates performance by less than 5%. In the benchmark *ct*, the simulator overestimates performance by nearly 67%. We attribute this difference to the inaccuracies in the memory system model in *tsim-proc*, which does not simulate the contention in the L2 cache banks and the on-chip network connecting the banks. In *rspeed*, the simulator underestimates performance by 19%. We attribute this difference to the inaccuracies in memory dependence prediction. The simulator encounters a pathological case where it observes a dependence violation for a load and therefore issues all subsequent instances of the same load conservatively, causing

Benchmark	% difference
conv	11
ct	66
dct8x8	5
matrix	4
sha	0
vadd	5
a2time01	0
autocor00	4
bezier02	18
rspeed01	-19
MEAN*	13

Table A.1: Percentage difference in execution cycles between the TRIPS hardware and *tsim-proc*. MEAN measures the average absolute difference in performance measurement.

a degradation in performance.

Appendix B

Improving Performance Using Critical Path Analysis

In this section, we show the utility of critical path analysis described in Chapter 5 for improving the performance of an application. We use the program *memset* for this exercise. This program is a C library routine that sets a range of bytes in memory to a given value. We start with a previously hand-tuned version of *memset*. The optimizations performed by hand include aggressive hyperblock formation using loop unrolling and predication, and hand placement of instructions. These optimizations improve the performance of *memset* by over 8× compared to automatically compiled code. The rest of this section describes how the information from critical path analysis can improve the performance of *memset* further[1].

Table B.1 shows the breakdown of the critical path cycles for two versions of *memset*. We observe that nearly 70% of the critical path cycles in the baseline program were consumed by operand communication and instruction execution events. We further observe that a large fraction of these cycles are dynamic delays, which indicate contention stall cycles in the operand network links and the execution tile

[1]Doug Burger performed the various hand-optimizations for *memset*.

Event	Baseline			Optimized		
	SD	DD	TD	SD	DD	TD
BF	2728	12533	15261	4896	13144	18040
BC	0	181	181	0	1967	1967
BD	408	507	915	3968	5262	9230
OP	18730	17777	36507	9249	5839	15088
IE	9712	15554	25266	3871	1826	5697
RR	92	23	115	528	14314	14842
SF	0	0	0	0	0	0
RF	7678	79	7757	2	0	2
IF	2521	0	2521	5351	0	5351
LD	246	542	788	244	542	786
Total	42115	47196	89311	28109	42894	71003

Table B.1: Overall critical path breakdown for two versions of *memset*. Baseline refers to the original hand-optimized version. Optimized refers to the version after applying the optimizations guided by critical path analysis. The label SD denotes the static delay, DD denotes the dynamic delay, and TD denotes the total delay for an event. We refer the reader to Table 5.7 for a description of the critical path events.

issue slots. To identify the specific program block causing these contention cycles, we re-performed the analysis to track the critical path composition on a per-block basis. We observed that nearly 70% of the critical path cycles resulted from events in one program block `memset_test$6`. Table B.2 shows these results. Nearly 90% of all operand communication and instruction execution latencies in the program's overall critical path result from this block.

Instructions in the block `memset_test$6` belong to one of four categories: store instructions, move instructions to distribute the base address for the stores, move instructions to distribute the data for the stores, and loop induction instructions. To identify specific instructions that cause bottlenecks, we re-ran the analysis to obtain critical path breakdowns for each instruction in the block `memset_test$6`. Table B.3 shows the contribution of the top five instructions in that block to the critical operand communication and instruction execution latencies. We observe that nearly 50% of these cycles resulted from just one single instruction. Examining the tile placements in the schedule, we observed that this instruction, a store, was

Event	Baseline			Optimized		
	SD	DD	TD	SD	DD	TD
OP	15650	17276	32926	5755	4310	10065
IE	7733	15209	22942	2120	1181	3301
RF	7548	0	7548	0	0	0
RR	0	0	0	400	12355	12755
SF	0	0	0	0	0	0
IF	114	0	114	2312	0	2312
LD	0	0	0	0	0	0
BF	264	61	325	2432	580	3012
BC	0	129	129	0	1485	1485
BD	312	429	741	3200	4400	7600
Total (cycles)	31621	33104	64725	16219	24311	40530

Table B.2: Critical path composition for the block `memset_test$6` in the program *memset*.

obtaining its base address from a move instruction placed at a different execution tile. In fact, all the consuming stores of this move instruction were placed at a different tile than the move. This artifact introduced one cycle of operand communication latency between the issue of the move and the target store instructions. To remove this latency, we re-adjusted the schedules by placing the move instruction and all of its target stores at the same tile.

The results for the optimized version of `memset_test$6` are shown in Tables B.1, B.2, and B.3 under the label *Optimized*. We observe that the overall performance improved by nearly 11%. As expected, we observe a significant re-

Baseline			Optimized		
SD	DD	TD	SD	DD	TD
7506	25288	32794	1172	631	1803
7484	819	8303	879	101	980
95	304	399	424	473	897
93	156	249	289	578	867
128	85	213	586	71	657

Table B.3: Operand communication and instruction execution cycles on the critical path for the top five instructions in the block `memset_test$6` in the program *memset*.

duction in operand communication and instruction execution latencies on the program's overall critical path. The critical path contribution of the top-most block, still `memset_test$6`, decreased by a greater fraction than the overall execution time. This behavior occurs because portions of the execution paths through this block are no longer critical compared to concurrent paths through other blocks. Table B.1 also shows significant increases in the contributions of the block fetch, block completion, and block commit operations. The block `memset_test$6` also exhibits similar sharp increases in contributions of other events. Reducing the effect of operand communication and instruction execution bottlenecks exposes these new bottlenecks which are candidates for future optimizations.

Appendix C

Front-End Performance in the TRIPS Architecture

In Chapter 6, we reported the effect of the front-end architecture on the overall execution critical path in various benchmarks. In this section, we report the raw performance data for the I-cache miss rates and control flow prediction accuracy in various benchmarks. Tables C.1 and C.2 report these data obtained using simulation. The second column shows the miss rates in the primary instruction cache. It measures the percentage of committed blocks that were filled from the L2 before they could be fetched and executed in the processor. The third column shows the control flow misprediction rate. It shows the percentage of committed blocks whose addresses were incorrectly predicted by the next-block predictor.

Benchmark	I-cache miss rate (%)	Control prediction miss rate (%)
dct8x8	3.05	11.37
matrix	1.17	10.20
sha	7.15	3.84
vadd	1.00	1.24
a2time01	0.56	0.23
basefp01	0.71	0.32
rspeed01	0.54	2.59
tblook01	0.92	10.58
bezier02	0.55	1.55
autocor00	0.54	2.01
conv	5.76	4.33
ct	9.47	7.65
genalg	13.76	17.97

Table C.1: Front-end performance for hand-optimized benchmarks. Results were obtained using *tsim-proc* simulation.

Benchmark	I-cache miss rate (%)	Control prediction miss rate (%)
164.gzip	0.08	5.85
181.mcf	0.12	11.03
186.crafty	31.34	11.24
197.parser	2.61	6.04
255.vortex	22.37	6.31
256.bzip2	0.00	8.51
300.twolf	22.35	10.45
253.perlbmk	14.60	12.47
168.wupwise	0.04	0.04
171.swim	0.00	0.08
172.mgrid	0.06	0.49
173.applu	0.00	0.13
177.mesa	20.03	8.90
179.art	0.04	0.10
200.sixtrack	14.10	15.58
301.apsi	16.92	2.66

Table C.2: Front-end performance for SPEC benchmarks. Results were obtained using *tsim-proc* simulation.

Bibliography

[1] EEMBC: The embedded microprocessor benchmark consortium. `http://www.eembc.org`.

[2] International technology roadmap for semiconductors 2006 update: Process integration, devices, and structures. `http://www.itrs.net`.

[3] Scale: A scalable compiler for a moving target. `http://www-ali.cs.umass.edu/Scale`.

[4] Trimaran : An infrastructure for research in instruction-level parallelism. `http://www.trimaran.org`.

[5] Carmelo Acosta, Sriram Vajapeyam, Alex Ramirez, and Mateo Valero. CDE: A Compiler-driven, Dependence-Centric, Eager-executing Architecture for the Billion Transistors Era. In *Proceedings of the 2003 International Workshop on Complexity-Effective Design*, June 2003.

[6] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[7] H. Akkary, R. Rajwar, and S.T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of*

the 36th Annual IEEE/ACM International Symposium on Microarchitecture, pages 423–434, December 2003.

[8] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th Annual Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[9] Donald Alpert and Dror Avnon. Architecture of the Pentium microprocessor. *IEEE Micro*, 13(3):11–21, 1993.

[10] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 Model 91: Machine philosophy and instruction handling. *IBM Journal of Research and Development*, 11(1):8–24, January 1967.

[11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[12] Tom Asprey, Gregory S. Averill, Eric DeLano, Russ Mason, Bill Weiner, and Jeff Yetter. Performance Features of the PA7100 Microprocessor. *IEEE Micro*, 13(3):22–35, 1993.

[13] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351, May 1992.

[14] R. Iris Bahar and Srilatha Manne. Power and energy reduction via pipeline

balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 218–229, June 2001.

[15] Saisanthosh Balakrishnan and Gurindar S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *Proceedings of the 33th Annual International Symposium on Computer Architecture*, pages 302–313, June 2006.

[16] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Dynamically allocating processor resources between nearby and distant ILP. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 26–37, June 2001.

[17] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 237–248, December 2001.

[18] Max Baron. Low-Key Intel 80-Core Intro: The Tip of the Iceberg. *Microprocessor Report*, April 2007.

[19] Eric Borch, Srilatha Manne, Joel Emer, and Eric Tune. Loose loops sink chips. In *Proceedings of the Eighth International Symposium on High Performance Computer Architecture*, pages 299–310, February 2002.

[20] Edward Brekelbaum, Jeff Rupley, Chris Wilkerson, and Bryan Black. Hierarchical scheduling windows. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 27–36, November 2002.

[21] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of*

the 27th Annual International Symposium on Computer Architecture, pages 83–94, June 2000.

[22] Mary D. Brown, Jared Stark, and Yale N. Patt. Select-free instruction scheduling logic. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 204–213, December 2001.

[23] Mihai Budiu, Pedro V. Artigas, and Seth Copen Goldstein. Dataflow: A complement to superscalar. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177–187, March 2005.

[24] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 14–26, October 2004.

[25] J. Adam Butts and Gurindar S. Sohi. Use-based register caching with decoupled indexing. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 302–313, June 2004.

[26] Alper Buyuktosunoglu, David H. Albonesi, Pradip Bose, Peter W. Cook, and Stanley E. Schuster. Tradeoffs in power-efficient issue queue design. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, pages 184–189, August 2002.

[27] Harold W. Cain and Mikko H. Lipasti. Memory ordering: A value-based approach. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 90–101, June 2004.

[28] Brad Calder, Glenn Reinman, and Dean M. Tullsen. Selective value predic-

tion. In *Proceedings of the 26th Annual International Symposium on Computer architecture*, pages 64–74, May 1999.

[29] Ramon Canal and Antonio Gonzalez. A low-complexity issue logic. In *Proceedings of the 14th International Conference on Supercomputing*, pages 327–335, May 2000.

[30] Ramon Canal and Antonio Gonzalez. Reducing the complexity of the issue logic. In *Proceedings of the 15th International Conference on Supercomputing*, pages 312–320, June 2001.

[31] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The ATOMOS transactional programming language. In *Proceedings of the 2006 Conference on Programming Language Design and Implementation*, pages 1–13, June 2006.

[32] Shailender Chaudhry, Paul Caprioli, Sherman Yip, and Marc Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, 2005.

[33] David Chinnery and Kurt Keutzer. *Closing the Gap Between ASIC & Custom Tools and Techniques for High-Performance ASIC Design*. Kluwer Academic Publishers, 2002.

[34] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 142–153, June 1998.

[35] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 306–317, December 2001.

[36] Thomas M. Conte, Sanjeev Banerjia, Sergei Y. Larin, Kishore N. Menezes, and Sumedh W. Sathaye. Instruction fetch mechanisms for VLIW architectures with compressed encodings. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 201–211, December 1996.

[37] Thomas M. Conte and Sumedh W. Sathaye. Dynamic rescheduling: A technique for object code compatibility in VLIW architectures. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 208–218, 1995.

[38] Katherine E. Coons, Xia Chen, Sundeep K. Kushwaha, Doug Burger, and Kathryn S. McKinley. A spatial path scheduling algorithm for EDGE architectures. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, October 2006.

[39] Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. Out-of-order commit processors. In *Proceedings of the Tenth International Symposium on High Performance Computer Architecture*, pages 48–59, February 2004.

[40] Jose-Lorenzo Cruz, Antonio Gonzalez, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 316–325, June 2000.

[41] William J. Dally and Brian Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proceedings of the 38th Design Automation Conference*, pages 684–689, June 2001.

[42] J. Dennis and D. Misunas. A preliminary architecture for a basic data-flow

processor. In *Proceedings of the 2nd Annual International Symposium on Computer Architecture*, pages 126–132, January 1975.

[43] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, June 2001.

[44] Keith Diefendorff. Compaq Chooses SMT for Alpha. *Microprocessor Report*, 13(16), December 1999.

[45] Keith Diefendorff. Power4 focuses on memory bandwidth. *Microprocessor Report*, 13(13), October 1999.

[46] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, March 2000.

[47] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing*, pages 68–75, June 1997.

[48] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Archtecture*, pages 26–37, June 1997.

[49] Dan Ernst, Andrew Hamel, and Todd Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 253–263, June 2003.

[50] Alexandre Farcy, Olivier Temam, Roger Espasa, and Toni Juan. Dataflow analysis of branch mispredictions and its application to early resolution of

branch outcomes. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 59–68, December 1998.

[51] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 149–159, December 1997.

[52] Brian Fields, Rastislav Bodik, and Mark D. Hill. Slack: Maximizing performance under technological constraints. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 47–58, June 2002.

[53] Brian Fields, Shai Rubin, and Rastislav Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[54] Brian A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 228–239, December 2003.

[55] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[56] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, June 1983.

[57] Joseph A. Fisher and B.R. Rau. Instruction-level parallel processing. *Science*, 253(5025):1233–1241, 1991.

[58] Brian R. Fisk and R. Iris Bahar. The non-critical buffer: Using load latency

tolerance to improve data cache efficiency. In *Proceedings of the 1999 IEEE International Conference on Computer Design*, pages 538–545, October 1999.

[59] Daniele Folegnani and Antonio Gonzalez. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 230–239, June 2001.

[60] Manoj Franklin and Mark Smotherman. A fill-unit approach to multiple instruction issue. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 162–171, November 1994.

[61] Alok Garg, M. Wasiur Rashid, and Michael Huang. Slackened memory dependence enforcement: Combining opportunistic forwarding with decoupled verification. In *Proceedings of the 33th Annual International Symposium on Computer Architecture*, pages 142–154, June 2006.

[62] Maria Jesus Garzaran, Milos Prvulovic, Jose Maria Llaberia, Victor Vinals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture*, pages 191–202, February 2003.

[63] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High Performance Computer Architecture*, pages 195–206, January 1998.

[64] Masahiro Goshima, Kengo Nishino, Toshiaki Kitamura, Yasuhiko Nakashima, Shinji Tomita, and Shin-Ichiro Mori. A high-speed dynamic instruction scheduling scheme for superscalar processors. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 225–236, December 2001.

[65] Paul Gratz, Changkyu Kim, Robert McDonald, Stephen W. Keckler, and Doug Burger. Implementation and Evaluation of On-Chip Network Architectures. In *Proceedings of the 2006 IEEE International Conference on Computer Design*, October 2006.

[66] Paul Gratz, Karthikeyan Sankaralingam, Heather Hanson, Premkishore Shivakumar, Robert McDonald, Stephen W. Keckler, and Doug Burger. Implementation and Evaluation of Dynamically Routed Processor Operand Network. In *To Appear in the 1st ACM/IEEE International Symposium on Networks-on-Chip*, May 2007.

[67] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[68] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, June 2004.

[69] Allan Hartstein and Thomas R. Puzak. Optimum power/performance pipeline depth. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 117–128, December 2003.

[70] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., fourth edition, 2006.

[71] John L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[72] Dana S. Henry, Bradley C. Kuszmaul, Gabriel H. Loh, and Rahul Sami. Circuits for wide-window superscalar processors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 236–247, June 2000.

[73] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 1, February 2001.

[74] M.S. Hrishikesh, Keith Farkas, Norman P. Jouppi, Doug Burger, Stephen W. Keckler, and Premkishore Sivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 14–24, May 2002.

[75] Michael Huang, Jose Renau, and Josep Torrellas. Energy-efficient hybrid wakeup logic. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, pages 196–201, August 2002.

[76] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, and Rumi Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5):12–23, September/October 2000.

[77] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, 1993.

[78] Daniel A. Jimenez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd annual*

ACM/IEEE International Symposium on Microarchitecture, pages 67–76, December 2000.

[79] R.E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[80] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[81] Ilhyun Kim and Mikko H. Lipasti. Half-price architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 28–38, June 2003.

[82] Nam Sung Kim and Trevor Mudge. Reducing register ports using delayed write-back queues and operand pre-fetch. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 172–182, June 2003.

[83] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 52–63, June 2004.

[84] Kevin Krewell. Sun's Niagara pours on the cores. *Microprocessor Report*, 18(9):11–13, September 2004.

[85] Kevin Krewell. Intel looks to Core for success. *Microprocessor Report*, March 2006.

[86] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[87] Gurhan Kucuk, Kanad Ghose, Dimitry V. Ponomarev, and Peter M. Kogge. Energy-efficient instruction dispatch buffer design for superscalar processors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 237–242, August 2001.

[88] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 59–70, June 2002.

[89] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, October 1998.

[90] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution on simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, June 2001.

[91] Bertrand A. Maher, Aaron Smith, Doug Burger, and Kathryn S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76, December 2006.

[92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, December 1992.

[93] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart Memories: A modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 161–171, June 2000.

[94] Jose F. Martinez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 3–14, November 2002.

[95] R. McDonald, Doug Burger, Stephen W. Keckler, K. Sankaralingam, and R. Nagarajan. TRIPS Processor Reference Manual. Technical report, Department of Computer Sciences, The University of Texas at Austin, 2005. `http://www.cs.utexas.edu/~trips`.

[96] Cameron McNairy and Don Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, March/April 2003.

[97] Stephen Melvin and Yale Patt. Enhancing instruction scheduling with a block-structured ISA. *International Journal of Parallel Programming*, 23(3):221–243, 1995.

[98] Stephen W. Melvin, Michael Shebanow, and Yale N. Patt. Hardware support for large atomic units in dynamically scheduled machines. In *Proceedings of the 21st Annual Workshop and Symposium on Microprogramming and Microarchitecture*, pages 60–63, November 1988.

[99] Pierre Michaud and Andre Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, pages 27–36, January 2001.

[100] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Mihai Budiu, and Seth C. Goldstein. Tartan: Evaluating spatial computation for whole program execution. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 163–174, October 2006.

[101] Teresa Monreal, Antonio Gonzalez, Mateo Valero, Jose Gonzalez, and Victor Vinals. Delaying physical register allocation through virtual-physical registers. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 186–192, November 1999.

[102] Charles Moore. Keynote Talk: Managing the Transition from Complexity to Elegance, 2003 International Workshop on Complexity-Effective Design. `http://www.ece.rochester.edu/~albonesi/wced03/slides/moore.pdf`.

[103] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[104] Enric Morancho, Jose Maria Llaberia, and Angel Olive. Recovery mechanism for latency misprediction. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, pages 118–128, September 2001.

[105] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar 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.

[106] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order

processors. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture*, pages 129–140, February 2003.

[107] Ramadass Nagarajan, Xia Chen, Robert G. McDonald, Doug Burger, and Stephen W. Keckler. Critical path analysis of the TRIPS architecture. In *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 37–47, March 2006.

[108] Ramadass Nagarajan, Sundeep K. Kushwaha, Doug Burger, Kathryn S. McKinley, Calvin Lin, and Stephen W. Keckler. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 74–84, October 2004.

[109] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 40–51, December 2001.

[110] A. Nicolau and J. Fisher. Measuring the parallelism available for very long word architectures. *IEEE Transactions on Computers*, 33(11):968–974, November 1984.

[111] John Oliver, Ravishankar Rao, Paul Sultana, Jedidiah R. Crandall, Erik Czernikowski, Leslie W. Jones IV, Diana Franklin, Venkatesh Akella, and Frederic T. Chong. Synchroscalar: A multiple clock domain, power-aware, tile-based embedded processor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 150–161, June 2004.

[112] S. Onder and R. Gupta. Superscalar execution with direct data forwarding.

In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 130–135, October 1998.

[113] Chong-Liang Ooi, Seon Wook Kim, Il Park, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Proceedings of the 15th International Conference on Supercomputing*, pages 368–380, June 2001.

[114] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 303–312, October 1999.

[115] Subbarao Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, Department of Computer Sciences, University Of Wisconsin Madison, 1998.

[116] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Archtecture*, pages 206–218, June 1997.

[117] David B. Papworth. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, 16(2):8–15, 1996.

[118] Il Park, Michael D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 171–182, November 2002.

[119] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[120] Andrew Petersen, Andrew Putnam, Martha Mercaldi, Andrew Schwerin, Susan Eggers, Steve Swanson, and Mark Oskin. Reducing control overhead in dataflow architectures. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 182–191, September 2006.

[121] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *International Solid-State Circuits Conference*, pages 184–185, February 2005.

[122] Zachary Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A study of slipstream processors. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 269–280, December 2000.

[123] Steven E. Raasch, Nathan L. Binkert, and Steven K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 318–329, May 2002.

[124] Ryan Rakvic, Bryan Black, Deepak Limaye, and John Paul Shen. Non-vital loads. In *Proceedings of the Eighth International Symposium on High Performance Computer Architecture*, pages 165–174, February 2002.

[125] Nitya Ranganathan. Control flow speculation for distributed architectures, Ph.D proposal, April 2007.

[126] Nitya Ranganathan, Ramadass Nagarajan, Daniel A. Jimenez, Doug Burger, Stephen W. Keckler, and Calvin Lin. Combining hyperblocks and exit predic-

tion to increase front-end bandwidth and performance. Technical Report TR-02-41, Department of Computer Sciences, The University of Texas at Austin, September 2002.

[127] B. Ramakrishna Rau. Dynamically scheduled VLIW processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 80–92, November 1993.

[128] Glenn Reinman, Todd M. Austin, and Brad Calder. A scalable front-end architecture for fast instruction delivery. In *Proceedings of 26th Annual International Symposium on Computer Architecture*, pages 234–245, May 1999.

[129] E.M. Riseman and C.C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, 21(12):1405–1411, December 1972.

[130] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 138–148, December 1997.

[131] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, pages 37–50, January 2001.

[132] Karthikeyan Sankaralingam. *Polymorphous Architectures: A Unified Approach for Extracting Concurrency of Different Granularities*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, October 2006.

[133] Karthikeyan Sankaralingam, Stephen W. Keckler, William R. Mark, and Doug Burger. Universal mechanisms for data-parallel architectures. In *Proceedings*

of the 36th Annual International Symposium on Microarchitecture, pages 303–314, December 2003.

[134] Karthikeyan Sankaralingam, Ramadass Nagarajan, Doug Burger, and Stephen W. Keckler. A technology-scalable architecture for fast clocks and high ILP. In Gyungho Lee and Pen-Chung Yew, editors, *Interaction between Compilers and Computer Architectures*, volume 619 of *The International Series in Engineering and Computer Science*, pages 117–139. Kluwer Academic Publishers, 2001.

[135] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[136] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M.S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 480–491, December 2006.

[137] Karthikeyan Sankaralingam, Vincent Ajay Singh, Stephen W. Keckler, and Doug Burger. Routed inter-ALU networks for ILP scalability and performance. In *Proceedings of the 2003 IEEE International Conference on Computer Design*, pages 170–179, October 2003.

[138] John S. Seng, Eric S. Tune, and Dean M. Tullsen. Reducing power with dynamic critical path information. In *Proceedings of the 34th Annual ACM/IEEE*

International Symposium on Microarchitecture, pages 114–123, December 2001.

[139] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable memory disambiguation for high ILP processors. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 399–410, December 2003.

[140] Simha Sethumadhavan, Robert McDonald Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Design and implementation of the TRIPS primary memory system. In *Proceedings of the 2006 IEEE International Conference on Computer Design*, 2006.

[141] Simha Sethumadhavan, Franziska Roesner, Doug Burger, Stephen W. Keckler, and Joel Emer. Late-binding: Enabling unordered load-store queues. In *To Appear in the 34th International Symposium on Computer Architecture*, June 2007.

[142] Andre Seznec, Eric Toullec, and Olivier Rochecouste. Register write specialization register read specialization: A path to complexity-effective wide-issue superscalar processors. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 383–394, November 2002.

[143] Tingting Sha, Milo M. K. Martin, and Amir Roth. Scalable store-load forwarding via store queue index prediction. In *Proceedings of the 38th Annual International Symposium on Microarchitecture*, pages 159–170, November 2005.

[144] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, September 2001.

[145] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[146] Vincent Ajay Singh, Karthikeyan Sankaralingam, Stephen W. Keckler, and Doug Burger. Design and Analysis of Routed Inter-ALU Networks for ILP Scalability and Performance. Technical Report TR2003-17, Department of Computer Sciences, The University of Texas at Austin, July 2003.

[147] Aaron Smith, Jim Burrill, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, and Kathryn S. McKinley. Compiling for EDGE architectures. In *Fourth International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, pages 185–195, March 2006.

[148] Aaron Smith, Jon Gibson, Jim Burrill, Robert McDonald, Doug Burger, Stephen W. Keckler, and Kathryn S. McKinley. TRIPS intermediate language (TIL) manual. Technical Report TR-05-20, Department of Computer Sciences, The University of Texas at Austin, March 2005.

[149] Aaron Smith, Ramadass Nagarajan, Karthikeyan Sankaralingam, Robert McDonald, Doug Burger, Stephen W. Keckler, and Kathryn S. McKinley. Dataflow predication. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–100, December 2006.

[150] J.E. Smith and G.S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, December 1995.

[151] G. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptible pipelined processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34, June 1987.

[152] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[153] Gurindar S. Sohi and Amir Roth. Speculative multithreaded processors. *Computer*, 34(4):66–73, 2001.

[154] Srikanth T. Srinivasan, Roy Dz ching Ju, Alvin R. Lebeck, and Chris Wilkerson. Locality vs. criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 132–143, July 2001.

[155] Srikanth T. Srinivasan and Alvin R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 148–159, November 1998.

[156] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, October 2004.

[157] Viji Srinivasan, David Brooks, Michael Gschwind, Pradip Bose, Victor Zyuban, Philip N. Strenski, and Philip G. Emma. Optimizing pipelines for power and performance. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 333–344, November 2002.

[158] Jared Stark, Mary D. Brown, and Yale N. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 57–66, December 2000.

[159] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the*

27th Annual International Symposium on Computer Architecture, pages 1–12, June 2000.

[160] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[161] Samantika Subramaniam and Gabriel H. Loh. Fire-and-forget: Load/store scheduling with no store queue at all. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 273–284, December 2006.

[162] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 291–302, December 2003.

[163] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[164] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture*, pages 341–353, February 2003.

[165] J. E. Thornton. Parallel Operation in the Control Data 6600. In *AFIPS Conference Proceedings, 1964 Fall Joint Computer Conference*, pages 33–41, 1964.

[166] G.S. Tjaden and M.J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, 19(10):889–895, October 1970.

[167] Marc Tremblay. Multithreaded multicores, an update from Sun. General-Purpose GPU Computing: Practice And Experience, Supercomputing 2006 Workshop. `http://www.gpgpu.org/sc2006/workshop/presentations/Tremblay_SC06.pdf`.

[168] Jessica H. Tseng and Krste Asanovic. Banked multiported register files for high-frequency superscalar microprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 62–71, June 2003.

[169] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[170] Eric Tune, Dongning Liang, Dean M. Tullsen, and Brad Calder. Dynamic prediction of critical path instructions. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, pages 185–195, January 2001.

[171] Eric Tune, Dean M. Tullsen, and Brad Calder. Quantifying instruction criticality. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 104–113, September 2002.

[172] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: RAW machines. *IEEE Computer*, 30(9):86–93, September 1997.

[173] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.

[174] Steven Wallace and Nader Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, pages 179–185, October 1996.

[175] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 51–61, 1991.

[176] Robert Yung and Neil C. Wilhelm. Caching processor general registers. In *Proceedings of the 1995 IEEE International Conference on Computer Design*, pages 307–312, October 1995.

[177] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, pages 135–141, January 1999.

[178] Huiyang Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 231–242, October 2005.

[179] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, June 2001.

[180] Craig Zilles and Gurindar Sohi. Master/slave speculative parallelization. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 85–96, December 2001.

Vita

Ramadass Nagarajan was born in Neyveli, India, on October 9, 1977, the son of N. Jayam and V. Nagarajan. He graduated from the Jawahar Higher Secondary School in 1995 and subsequently enrolled at the Indian Institute of Technology, Madras, where he earned the Bachelor of Technology in Computer Sciences and Engineering. In the fall of 1999, he joined the doctoral program at the Department of Computer Sciences at the University of Texas at Austin. While pursuing his Ph.D degree, he obtained the degree of Master of Science in Computer Sciences in December, 2001.

Permanent Address: Flat G-2, Subasri Athreya,
55, Bhuvaneshwari Nagar, Chromepet,
Chennai - 600044, India.

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.