Copyright
by
Karthikeyan Sankaralingam
2007

The Dissertation Committee for Karthikeyan Sankaralingam
certifies that this is the approved version of the following dissertation:

Polymorphous Architectures: A Unified Approach for Extracting Concurrency of Different Granularities

Committee:

Stephen W. Keckler, Supervisor

Saman Amarasinghe

James C. Browne

Douglas C. Burger

H. Peter Hofstee

William R. Mark

Polymorphous Architectures: A Unified Approach for Extracting Concurrency of Different Granularities

by

Karthikeyan Sankaralingam, 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

August 2007

Acknowledgments

"At times our own light goes out and is rekindled by a spark from another person. Each of us has cause to think with deep gratitude of those who have lighted the flame within us."

–Albert Schweitzer

Many people have contributed to my dissertation research and my experience at UT. I would like to first thank my advisor, Steve Keckler, for his advice, guidance, and training that helped me get to this point in my graduate career. Thanks also for paying me during more than seven years of graduate school and bringing me to Austin. Steve Keckler and Doug Burger provided the vision, constant encouragement and technical expertise that made this dissertation possible. Steve has been an excellent mentor and thesis advisor and I cannot thank him enough for the influence he has had on me.

Doug Burger, as the co-leader of the CART group, and a de-facto co-advisor for my dissertation research has also played an important part in my professional development. I am thankful for the numerous opportunities I have had to interact with him, and for his many insightful comments on topics ranging from microarchitecture pipelines to brewing beer.

The students in the CART lab have been incredible and made graduate school life always exciting. I would like to thank them for their feedback on my research, attending many of my practice talks, proofreading my papers, and indulging in numerous technical and non-technical discussions that kept me entertained, informed, and always provoked my thinking. Thanks especially to Vikas Agarwal, Hrishi, Changkyu Kim, Kartik Agaram, Simha Sethumadhavan, Raj Desikan, and Heather Hanson for their inputs on all topics of research and life in general. Thanks to Maria Jump and Alison Norman for attending numerous practice talks and giving their perspective on my research. Thanks to Heather Hanson and Kartik Agaram for carefully proof-reading this document and riding me of my under-hyphenation disease.

I would especially like to thank Ramadass Nagarajan with whom I collaborated closely through all my years in graduate school. I have thoroughly enjoyed working together with him and could not have asked for a better collaborator. Starting from our initial work on writing a PowerPC port of the SimpleScalar simulator, to our late night brainstorming discussions on initial ideas for the TRIPS processor, and design and verification of the prototype chip, I have always enjoyed Ramdas's insights and hearing his perspective. We jointly designed the instruction set described in this dissertation and many aspects of the microarchitecture.

I would like to thank the staff in the Computer Sciences department for helping navigate graduate school bureaucracy and shielding me from most of it. The facilities staff in the department were outstanding, and the rare times they exposed their BOFH side to me were completely justified! I would like to thank Gem Naivar for helping with graduate school miscellany, several travel issues, and her immense patience and tolerance.

Last, but not the least, I would like thank my family. My mother and my brother provided me with the constant encouragement necessary for success in graduate school and I could not have done it without them. Thanks Mom for your infinite patience and never asking me when I was going to graduate! Thanks Ranga for putting up with me as a roommate for two years and tolerating all my eccentricities as a kid brother for 28 years. Thanks Dad for your words of wisdom and challenging me to be the best I could.

I would also like to acknowledge the institutions that helped support my research in graduate school: National Science Foundation for the initial grants that funded me and DARPA for the TRIPS funding.

Polymorphous Architectures: A Unified Approach for Extracting Concurrency of Different Granularities

Publication No. ____________________

Karthikeyan Sankaralingam, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Stephen W. Keckler

Processor architects today are faced by two daunting challenges: emerging applications with heterogeneous computation needs and technology limitations of power, wire delay, and process variation. Designing multiple application-specific processors or specialized architectures introduces design complexity, a software programmability problem, and reduces economies of scale. There is a pressing need for design methodologies that can provide support for heterogeneous applications, combat processor complexity, and achieve economies of scale. In this dissertation, we introduce the notion of *architectural polymorphism* to build such scalable processors that provide support for heterogeneous computation by supporting different granularities of parallelism. Polymorphism configures coarse-grained microarchitecture blocks to provide an adaptive and flexible processor substrate. Technology scalability is achieved by designing an architecture using scalable and modular microarchitecture blocks.

We use the dataflow graph as the unifying abstraction layer across three granularities of parallelism–instruction-level, thread-level, and data-level. To first order, this granularity of parallelism is the main difference between different classes of applications. All programs are expressed in terms of dataflow graphs and directly mapped to the hardware, appropriately partitioned as required by the granularity of parallelism. We introduce Explicit Data Graph Execution (EDGE) ISAs, a class of ISAs as an architectural solution for efficiently expressing parallelism for building technology scalable architectures.

We developed the TRIPS architecture implementating an EDGE ISA using a heavily partitioned and distributed microarchitecture to achieve technology scalability. The two most significant features of the TRIPS microarchitecture are its heavily partitioned and modular design, and the use of microarchitecture networks for communication across modules. We have also built a prototype TRIPS chip in 130nm ASIC technology composed of two processor cores and a distributed 1MB Non-Uniform Cache Access Architecture (NUCA) on-chip memory system.

Our performance results show that the TRIPS microarchitecture which provides a 16-issue machine with a 1024-entry instruction window can sustain good instruction-level parallelism. On a set of hand-optimized kernels IPCs in the range of 4 to 6 are seen, and on a set of benchmarks with ample data-level parallelism (DLP), compiler generated code produces IPCs in the range of 1 to 4. On the EEMBC and SPEC CPU2000 benchmarks we see IPCs in the range of 0.5 to 2.3. Comparing performance to the Alpha 21264, which is a high performance architecture tuned for ILP, TRIPS is up to 3.4 times better on the hand optimized kernels.

However, compiler generated binaries for the DLP, EEMBC, and SPEC CPU2000 benchmarks perform worse on TRIPS compared to an Alpha 21264. With more aggressive compiler optimization we expect the performance of the compiler produced binaries to improve.

The polymorphous mechanisms proposed in this dissertation are effective at exploiting thread-level parallelism and data-level parallelism. When executing four threads on a single processor, significantly high levels of processor utilization are seen; IPCs are in the range of 0.7 to 3.9 for an application mix consisting of EEMBC and SPEC CPU2000 workloads. When executing programs with DLP, the polymorphous mechanisms we propose provide harmonic mean speedups of 2.1X across a set of DLP workloads, compared to an execution model of extracting only ILP. Compared to specialized architectures, these mechanisms provide competitive performance using a single execution substrate.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last decade, programmable processors have proliferated into increasingly diverse application domains, producing distinct markets for desktop, network, server, scientific, graphics, and digital signal processors. While clearly providing application-specific performance improvements, these processors perform poorly on applications outside of their intended domain, primarily because they are tuned to exploit specific types and granularities of parallelism, and to some extent due to instruction set specialization. Emerging applications with heterogeneous computational requirements, such as image recognition and tracking or video databases, introduce the need for computation systems that can support such heterogeneous computation. Future systems can be heterogeneous at the hardware level, built using multiple domain-specific processors to support this application heterogeneity. They suffer from two problems: reduced economies of scale compared to a single general purpose design and design-time freezing of the processor mix and composition. These two problems motivate the need for a flexible or *polymorphous* processor design that can adapt to different application demands dynamically.

Along with this proliferation of programmable processors, the performance of general purpose processors has grown tremendously over the past two decades.

This improvement has come from deeper pipelines and faster transistors. Device integration has played a large role in improving processor performance as well, enabling large on-chip multi-megabyte caches, multiple floating point units on chip, and microarchitecture structures to improve performance. Due to technology limitations of wire delays [4], power [74], and process variation [25], performance improvement due to pipelining and faster transistors is likely to slow down. Device integration has already reached a point where conventional architectures are unable to utilize more on-chip transistors to extract more performance. As a result, performance growth in the future must come from extracting more concurrency from applications. Architectures must extract concurrency at all levels, including thread-level and coarse-grained data-level parallelism, and not rely on only fine-grained instruction-level parallelism. But conventional architectures are poor at extracting such different granularities of parallelism and furthermore rely primarily on large centralized structures like register files, rename tables, and predictors to extract concurrency. Due to the aforementioned technology limitations, scaling conventional designs which are monolithic and integrated to future technologies is infeasible. There is instead a desire for scalable and modular architectures.

Broadly, the two trends that processor architects face are: 1) emerging applications with heterogeneous computation needs, and 2) technology limitations of power, wire-delay, and process variation. There is a growing need for design methodologies that can achieve economies of scale, provide support for heterogeneous applications, and combat the processor complexity arising from these technology trends. In this dissertation, we introduce *polymorphism* to build such scalable processors that provide support for such heterogeneous computation. The key idea behind polymorphism is to configure coarse-grained microarchitecture blocks to provide an adaptive and flexible processor substrate. Technology scalability is achieved by a designing an architecture using scalable and modular microarchitecture blocks.

Another strategy for addressing technology constraints and diverse application demands is to build a heterogeneous chip, which contains multiple processing cores, each designed to run a distinct class of workloads effectively. The Tarantula processor is one example of integrated heterogeneity [48]. The two major downsides to this approach are increased hardware complexity, since there is little design reuse between the types of processors and poor resource utilization when the application mix contains a balance different than that ideally suited to the underlying heterogeneous hardware.

The intent of a polymorphous design instead is to build one or more homogeneous processors, thus mitigating the aforementioned complexity problem. The polymorphous nature of the processor cores allows the hardware to be configured to provide special purpose behavior on an application-by-application basis, thus adapting to a wide range of application classes. Since the hardware is constructed of homogeneous processor cores, the resource utilization problem found in heterogeneous systems, of mis-match between application mix and hardware capability does not arise since the hardware can be adapted at run-time to any application mix.

In this dissertation, we define architectural polymorphism and describe a core set of principles which we build upon to develop mechanisms to implement polymorphism. We describe the TRIPS architecture which is a technology scalable and partitioned design. The TRIPS ISA is one instance of a new class of ISAs called Explicit Data Graph Execution (EDGE) which we propose in this dissertation as an architectural solution to expressing concurrency to the hardware. The polymorphous mechanisms are described in the context of the TRIPS architecture. In the remainder of this chapter we provide a short overview of polymorphism, a summary of the TRIPS architecture, and conclude with a thesis statement and a description of contributions.

1.1 Principles of Polymorphism

We define **architectural polymorphism** as *the ability to modify the functionality of coarse-grained microarchitecture blocks at runtime, by changing control logic but leaving datapath and storage elements largely unmodified, to build a programmable architecture that can be specialized on an application-by-application basis*. The main principles of polymorphism are the following which are developed in detail through the remainder of this dissertation:

- Adaptivity across different granularities of parallelism.
- Economy of mechanisms so that different microarchitecture structures are used differently at different times, rather than application-specific structures.
- Reconfiguring coarse-grained blocks to provide different functionality instead of synthesizing fine-grained primitive components into blocks with different functionality, as done by FPGAs.

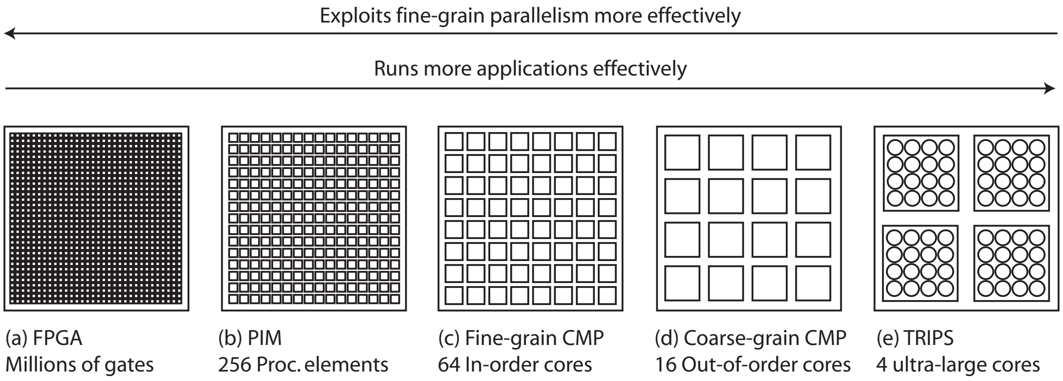


Figure 1.1: Granularity of parallel processing elements on a chip. Number of cores that can fit on a typical 65nm high performance chip.

1.2 System Design

Before applying this abstract definition of architectural polymorphism to processor architectures to develop the resources and mechanisms for implementing polymorphous systems, three main system decisions must be addressed: the granularity of processor cores, granularities of parallelism, and technology scalability.

1.2.1 Granularity of Processors

The granularity of processors spans the following spectrum shown in Figure 1.1.

a) Ultra-fine-grained FPGAs that consist of an array of gates or configurable lookup tables interconnected through a configurable network. These are typically programmed using a high-level hardware description language and applications are synthesized to the hardware.

b) Several basic processing cores like in PipeRench [59] or PACT-XPP [19]. The primitive processor elements provide more functionality than gates and lookup tables used in an FPGA. They are programmed at a higher level of abstraction than FPGAs and thus speed up the development process, however they still synthesize applications to hardware like an FPGA.

c) Many simple in-order processors like in the RAW architecture [156, 158] or Sun Niagara chip [92]. Each processing core is a full fledged processor that runs applications compiled down to the ISA of the processor. RAW also has the ability to use sophisticated compiler techniques to map a single application across these processing cores.

d) Many powerful out-of-order processors like in the POWER4 chip [159]. The processing cores are more powerful and provide higher single-thread performance than the above three.

e) Some number of ultra-wide issue processors like the Grid Processor [117]–a TRIPS chip like configuration we propose in this dissertation.

Fine-grained architectures perform well when ample fine-grained parallelism exists but do not support general purpose sequential programs. They are plagued by synchronization overheads resulting from aggregating multiple of these units together. Coarse-grained architectures using conventional wide-issue out-of-order processors have the ability for high performance on sequential codes, but have traditionally lacked the capability for partitioning and support for fine-grained parallelism. Technology limitations of power and wire delays limit the scalability of conventional out-of-order processor designs.

In this dissertation, we assert that a chip with few large cores is better than many fine grained cores across a spectrum of applications if the coarse-grained cores can be subdivided when fine-grain parallelism exists. Our two key insights are: 1) Use the dataflow graph as a basic level of abstraction to express concurrency to the hardware to eliminate the hardware's need to rediscover concurrency, and reduce the hardware overheads of instruction-level bookkeeping. 2) The full processor core is designed to exploit coarse-grained concurrency and we use polymorphism to subdivide resources to support fine-grained concurrency.

1.2.2 Granularity of Parallelism

To first order, classes of applications can be represented by different types of concurrency. Desktop, server, network processing, digital signal processing, etc. can all be classified into three categories of parallelism:

Instruction-level Parallelism (ILP): The predominant type of parallelism is among individual machine operations, such as memory loads, stores, and arithmetic operations. The operations are simple RISC-style operations and the system is handed a single program written with a sequential processor in mind [134].

Thread-level Parallelism (TLP): Parallelism between multiple threads of control, either from the same program or from different programs.

Data-level Parallelism (DLP): Parallelism across groups of data that have the same or similar operations applied to them. Several data operands share a single flow of control.

The differences between application domains includes several other features:

- Memory access patterns which include streaming-like regular or more irregular accesses typical of recursive data structures.
- Instruction mix.
- Types of arithmetic operation, namely fixed point or floating point.
- Energy efficiency and power consumption. Embedded workloads typically operate in the milli-watt regime, whereas server workloads operate in the 60W to 80W regime.

However, at an architecture level, granularity of parallelism is the main difference between different application domains.

These classes of concurrency are not mutually exclusive. In fact, it is common to extract some amount of ILP in traditional multithreaded workloads like database workloads. An example of simultaneously using TLP and DLP is found in the Cell processor, where multithreading is extensively used to partition work among eight Synergistic Processing Engines which are SIMD execution units used to extract DLP. In the remainder of this dissertation, we examine polymorphism and application heterogeneity in the context of these three types of parallelism. While ILP and TLP are well understood, the differences between programs with DLP is less well understood. In chapter 8 we undertake a comprehensive program characterization of data-level parallelism to analyze the behavior of these programs

1.2.3 Technology Scalability

Conventional microarchitectures traditionally rely on large centralized structures like register files, branch prediction tables, and rename tables to extract concurrency [4]. Increasing wire delays and the limits on pipeline depth from a performance and power perspective restrict the scalability of these architectures [4, 80, 73, 151, 74]. Consequently, technology limitations have driven a desire for scalability, modularity, reduced complexity, and energy efficiency in processor architectures. Polymorphism could potentially satisfy these requirements.

- **Scalability and Modularity:** The basic ideas behind polymorphism lead to the construction of scalable and reconfigurable modular blocks to support multiple application domains.

- **Complexity:** The economy of mechanisms that is central to architectural polymorphous inherently reduces complexity and makes the architecture scalable.

- **Energy efficiency:** By using a small set of mechanisms and adapting the processor to an application's needs, polymorphous architectures can be energy efficient for wide class of domains compared to general purpose programmable processors. However, it is not clear how close polymorphous systems can get to the energy efficiency of specialized processors.

1.3 TRIPS Architecture

In this dissertation, we develop a technology scalable architecture called TRIPS which uses a new dataflow encoding ISA to express concurrency more efficiently to the hardware. The hardware is implemented using a distributed microarchitecture that relies on well defined control and data networks for communication. One contribution of this dissertation is the specification and description of this scalable and distributed architecture. The mechanisms to implement polymorphism are developed in the context of this architecture. We chose this architecture as our baseline upon which to develop the mechanisms for polymorphism because this design already provides a scalable and modular starting point. The main features of the architecture are:

1. Dataflow dependences are encoded in the ISA to enable direct instruction-instruction communication and reduce the overheads of detecting and managing dependencies that conventional out-of-order processors must pay. This new class of ISAs called EDGE (Explicit Data Graph Execution) essentially brings dataflow to the ISA, without having to change programming models. Unlike VLIW architectures, the execution order of instructions is determined dynamically based on when operands arrive at instruction slots, thus relieving the compiler of the responsibility of determining the dynamic execution order.

2. The program is partitioned into well-defined blocks to limit the scope of the dependences so that the number of dependence arcs does not exceed the instruction space. Dependences inside such a block are encoded directly in the instructions, while dependences across blocks are expressed through architectural registers or store-load pairs. This execution model fetches, executes, and commits a full block of instructions atomically to reduce the overheads of instruction management like register renaming, dependence checking, and branch prediction. These overheads are amortized across many instructions, thus saving energy per executed instruction.

3. To manage design complexity and address wire-delay scaling, the computation core is completely distributed using well defined microarchitecture control and data networks with only nearest-neighbor links for communication. The use of such well defined networks reduces design complexity because the the communication and interaction between units is only through these networks, compared to bypass paths and stall signals as is common in conventional designs. Furthermore, the microarchitecture is constructed is using a set of small tiles such that these nearest-neighbor links can be traversed in a single cycle, and each tile's complexity is low.

1.4 Implementation of Polymorphism

Architectural polymorphism provides the capability to configure hardware at run-time to perform different functions. Unlike a reconfigurable architecture, a polymorphous architecture alters the behavior of coarse-grained components instead of synthesizing functions from primitive logic blocks at run-time.

Table 1.1 lists a taxonomy of high-level architectures principles used in processor design and defines the polymorphism approach using this taxonomy. The taxonomy provides a 4-tuple that can be used to classify architectures into one (or more) of 16 possible categories and polymorphous architectures occupy a portion of this space. In chapter 2 which discusses related work, we classify other architectures according to this taxonomy. Below, we briefly explain polymorphous architectures according to this taxonomy.

- **Architecture type:** Architecture type can be programmable hardware or application specific hardware. Programmable hardware refers to architectures that execute a program specified using an ISA that has been compiled into a program binary, with typically a small portion of the program's instructions mapped to execution resources on the hardware at one time. Application specific hardware on the other hand directly maps the functionality of the entire program into hardware elements like gates and data-path units with the full program mapped to the hardware at once. Programmability differentiates architectural polymorphism from other approaches to reconfiguration like FPGAs which create application specific hardware. Polymorphous architectures tailor a programmable architecture to application needs.

- **Processor type:** The processor cores used to construct a chip can be homogeneous or heterogeneous. While polymorphism does not require or imply a chip made of homogeneous processor cores, in this dissertation we restrict ourself to discussing and evaluating polymorphism for homogeneous cores. The Smart Memories chip is another example of a homogeneous polymorphous architecture.

- **Core granularity:** Core granularity can be coarse-grained or fine-grained, and we define a core as the set of units on-chip controlled by a single program counter. Architectural polymorphism can be implemented on fine-grained cores like simple in-order processors or coarse-grained cores like the TRIPS core. Designing polymorphous mechanisms for aggregating fine-grained cores to execute a large program presents different challenges from partitioning a coarse-grained core for supporting fine-grained concurrency. While aggregation introduces the challenge of overcoming synchronization overheads when multiple cores must communicate, for coarse-grained cores the challenge is efficiently partitioning the substrate to a sufficiently small level of granularity to support fine-grained parallelism.

- **Configuration granularity:** Architectural polymorphism is defined as configuration of coarse-grained microarchitecture blocks and is different from synthesizing different functions from fine-grained primitive components like datapath slices, like and FPGA, or primitive processing elements.

In this dissertation, we discuss polymorphism in the context of the TRIPS processor to support different granularities of parallelism. The main polymorphous resources in the TRIPS processor are: the *instruction window space, physical register files*, the *block sequencing logic*, and the *on chip memory system*.

<table>
<tr><th>Architecture type</th><th>Processor type</th><th>Core granularity</th><th>Configuration granularity</th></tr>
<tr><td>Programmable h/w</td><td>Homogeneous</td><td>Coarse-grained</td><td>Coarse-grained</td></tr>
<tr><td>Application specific h/w</td><td>Heterogeneous</td><td>Fine-grained</td><td>Fine-grained</td></tr>
<tr><td colspan="4"><b>Polymorphous Architectures</b></td></tr>
<tr><td>Programmable h/w</td><td>Homogeneous
or
Heterogeneous</td><td>Coarse-grained
or
Fine-grained</td><td>Coarse-grained</td></tr>
</table>

Table 1.1: A taxonomy of architectures.

While the concept and the mechanisms are explained in detail in Chapter 5 we briefly summarize the resources and provide some examples of polymorphism below. Using polymorphism the reservation stations can be reconfigured in the following ways to adapt the processor to different granularities of parallelism: 1) configure the reservation stations like an instruction window and devote all entries to one thread to support ILP, 2) share the reservation stations among multiple threads for TLP, and 3) provide instruction sequencing support at every ALU site to support fine-grained DLP that is best executed in a MIMD style of computation.

1.5 Thesis Statement

This dissertation introduces the concept of architectural polymorphism – the capability to configure coarse-grained microarchitecture blocks to provide application controlled specialization of an architecture. This dissertation presents the design and implementation of a scalable processor that can be configured to support different granularities of parallelism using polymorphous mechanisms. Specifically, this dissertation describes the TRIPS architecture and evaluates polymorphous mechanisms for supporting different granularities of parallelism on the TRIPS processor.

1.6 Dissertation Contributions

This dissertation makes the following main contributions.

Architectural Polymorphism: We introduce the concept of architectural polymorphism and develop the main principles and a set of mechanisms driven by these principles that configure coarse-grained microarchitecture blocks to support different granularities of parallelism. Compared to reconfigurable architectures which attempt to provide support for diverse workloads using a synthesis approach of building different functional blocks from primitive components, the principle behind polymorphism is to adapt coarse-grained blocks to behave differently.

TRIPS Architecture: We describe the TRIPS processor organization, its ISA (one instance of an EDGE ISA), and microarchitecture[1]. EDGE ISAs succinctly express concurrency to the hardware by encoding programs as sequences of atomic blocks of execution with blocks encoding a dataflow graph that can be directly mapped to physical resources in the processor. The TRIPS processor core provides a 1024-entry instruction window and can issue up to 16 instructions every cycle. We have also built a prototype chip in 130nm ASIC technology composed of two

[1] The principles behind EDGE ISAs and the implementation of the TRIPS ISA and its microarchitecture are not sole individual contributions but are collaboratory efforts in which I have played lead intellectual roles.

TRIPS processor cores and a distributed 1MB on-chip memory system which can be configured as a non-uniform cache architecture (NUCA).

Data-Parallel Program Attributes: We present a detailed characterization of the fundamental behavior of data-parallel programs based on their memory access patterns, program control behavior, and available concurrency.

Experimental Evaluation: Our performance results show that the TRIPS microarchitecture can sustain good instruction-level parallelism. On a set of hand-optimized kernels IPCs in the range of 4 to 6 are seen, and on a set of highly data-parallel benchmarks with compiler generated code IPCs in the range of 1 to 4 are seen. On the EEMBC and SPEC CPU2000 benchmarks we see IPCs in the range of 0.5 to 2.3. Comparing performance to the Alpha 21264, which is a high performance architecture tuned for ILP, TRIPS is up to 3.4 times better on the hand optimized kernels. However, the compiler generated binaries for the DLP, EEMBC, and SPEC CPU2000 benchmarks perform worse on TRIPS compared to an Alpha 21264. With more aggressive compiler optimization we expect the performance of the compiler produced binaries to improve.

With more aggressive compiler optimization we expect these numbers to improve.

The polymorphous mechanisms proposed in this dissertation are effective at exploiting thread-level parallelism and data-level parallelism. When executing 4 threads on a single processor, high levels of processor utilization are seen, IPCs are in the range of 0.7 to 3.9 for an application mix consisting of EEMBC and SPEC CPU2000 workloads. When executing programs with DLP, the polymorphous mechanisms we propose provide harmonic mean speedups of 2.1X across a set of DLP workloads, compared to an execution model of extracting only ILP. Compared to specialized architectures, these mechanisms provide competitive performance using a single execution substrate.

1.7 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 discusses related work and places this dissertation in the context of prior work. Chapter 3 defines and describes EDGE ISAs and the compilation strategy for this new class of ISAs. Chapter 4 describes the TRIPS architecture and the prototype TRIPS chip. We describe the TRIPS ISA, the microarchitecture of the TRIPS chip, and briefly describe the logic design, verification, synthesis and physical design of the prototype TRIPS chip.

Chapter 5 describes architectural polymorphism. We describe the three principles behind polymorphism and a classification scheme for processor resources into fixed, specialized, and polymorphous resources. We then describe the mechanisms and resources required to implement polymorphism to support ILP, TLP, and DLP in the TRIPS architecture.

Chapter 6 presents a performance evaluation of the TRIPS processor focused on instruction-level parallelism. The performance evaluation is based on an event driven validated processor simulator. Chapter 7 presents a performance evaluation of using polymorphous mechanisms in the TRIPS processor to extract thread-level parallelism.

Chapter 8 presents a detailed application characterization of data parallel programs based on their fundamental behavior. Based on this characterization a set of microarchitecture mechanisms to support data-level parallelism is proposed. This chapter also includes a performance evaluation of these mechanisms on a high-level processor simulator that models the TRIPS processor. Finally, chapter 9 concludes and points to some future directions in the software aspects of polymorphous systems and the application of polymorphism to optimize other technology constraints like power and area.

Chapter 2

Related Work

This chapter discusses and differentiates prior work most closely related to the focus of this dissertation. The related work is grouped around the four main themes of this dissertation: polymorphism, data parallel architectures, scalable architectures, and microarchitecture techniques for ILP.

2.1 Polymorphism

Below we discuss the previous work related to polymorphism. We discuss prior work that has focused on support for different types of applications on a single substrate using reconfiguration or other means.

Multithreading: While multithreading is not directly related to supporting different types of applications, polymorphism-like behavior has been used to support multithreading in modern processor. We briefly trace the history of multithreading before describing these systems. Multithreading has been widely used to share compute resources between multiple program threads [102]. Multithreaded pipelining was used in the Peripheral and Control Processors of the Control Data 6600 computer architecture of the early 1960s to provide several virtual peripheral processors [160]. More recently, the HEP multiprocessor system had limited polymorphous behavior. It included support for multiple program contexts in the processor and “it allowed the user to control the number of processes dynamically in order to take advantage of varying amounts of parallelism in a problem [148].” Other recent systems that provided multithreading support on a single chip include the MIT M-Machine [53], MIT Alewife machine [3], Hydra [70], and the Pirañha multiprocessor [18].

Fine-grained multithreading to share processor resources between threads has been explored using different techniques. The Tera computer system had support for fine-grained multithreading interleaving long instruction word (LIW) instructions from different threads every cycle [8]. Keckler and Dally proposed an architecture that incorporated both compile-time and run-time information to interleave multiple VLIW instructions on individual functional units [87]. Both of these have a polymorphous nature in the sense that they support single-thread execution and multiple threads using the same set of mechanisms. Tullsen et al. described their approach of supporting multiple thread contexts in the pipeline of a modern out-of-order processor and called it simultaneous multithreading (SMT) [164]. They method replicates certain architectural storage elements in the processor, but shares most other resources to support the execution of multiple threads simultaneously in the processor pipeline. Yamamoto and Nemirovsky proposed an architecture similar to SMT but with separate instruction queues for each thread [173]. Ungerer et al. provide a detailed survey of multithreading literature [166].

Novel architectures: Browne et al. developed the Texas Reconfigurable Array Computer that could support sequential processing, SIMD, and MIMD processing on a single substrate [83, 144]. The TRAC project was focused on building interconnection networks and optimizing communication for a configurable array that relied on large amounts of off-chip communication.

The Stanford Smart Memories project employs polymorphous mechanisms to synthesize a large core from a modular homogeneous substrate [107]. While this approach works well for thread-level and data-level parallelism, single threaded execution suffers on this architecture. The main conceptual difference between Smart Memories and TRIPS is that TRIPS has a well defined set of specialized resources and fixed resources that can be used to support specific application needs. For example, TRIPS has a traditional 2-way set associative instruction cache which provides high instruction fetch bandwidth and low latency instruction fetch. Its function does not change with application behavior. A second example is the next-block predictor used in TRIPS, which is used to predict control flow for sequential programs. In Smart Memories on the other hand, there are no such fixed resources like the instruction cache or specialized resources like the next-block predictor. Instead the architecture simply provides an array of tiles, with each tile containing multiple SRAM banks, an interconnection network, and a simple processor core. Synthesizing efficient instruction cache behavior out of these SRAM banks can be challenging and creating branch predictor-like behavior out of the memory tiles is almost impossible. While more homogeneous and perhaps simpler than the TRIPS design, the lack of any specialized resources makes this architecture less adaptable.

The Vector-Thread Architecture supports data parallel and multithreaded execution by configuring the instruction sequencing logic of a set of closely coupled processor cores [95]. This architecture provides a scalable, tightly integrated MIMD array for data intensive processing. Clearly it can excel on vector codes and fine-grained MIMD parallelism. However, this architecture lacks many mechanisms that are required for extracting ILP. For example, it lacks memory ordering mechanisms for load/store re-ordering. As a result it is unclear how well this architecture will perform on general purpose programs.

Sasanka et al. propose a novel architecture called ALP to support ILP, TLP, and DLP for media applications [139]. They introduce a DLP technique called SIMD vectors and streams (SVectors/SStreams), which is integrated within a conventional superscalar based CMP/SMT architecture with sub-word SIMD parallelism. The technique exploits the simple implementation of sub-word SIMD already common in many machines and provides the benefits of full-fledged vector processing. The primary focus of ALP is to support multiple types of parallelism on conventional architectures with evolutionary changes to the ISA and microarchitecture. Its main drawback is that it augments a conventional processor core and as a result it does not scale to large issue widths. The techniques proposed in ALP extend a conventional processor core to support parallelism efficiently, but do not address the wire-delay and complexity issues that plague scaling of the underlying microarchitecture. As a result, large amounts of DLP will have to be partitioned into threads and distributed across a set of narrow-issue cores. TRIPS on the other hand provides a scalable very wide-issue design that can be tailored to application needs using polymorphism.

Finally, Rabbah et al. introduce a *versatility* metric to quantify the ability of an architecture to effectively execute a broad set of applications [130]. They also propose a benchmark suite called VersaBench suite that is comprised of a set of applications that capture diverse behavior. This versatility metric is simply a quantitative metric for comparing different types of architectures and does not describe or characterize the architecture itself. They formally define versatility as: "the geometric mean of the speedup of each of the applications in the VersaBench suite relative to the architecture which provides the best execution time for that application."

Extensions to conventional designs: In addition to reconfiguration for performance, adaptivity has been used to increase energy efficiency. Albonesi et al. [7] introduce *adaptive processing* where on-chip structures are dynamically resized to provide power efficient execution. This can be thought of as polymorphism within the ILP domain that uses run-time application behavior to improve energy efficiency. Other examples of specific microarchitecture mechanisms to provide adaptability include the following: adjusting cache size via ways [6], sizing issue windows [56], adjusting the issue window coupled with the load/store queue and register file [127], adjusting issue width along with the functional units [14], and adaptively resizing instruction issue queues [80, 129].

At a coarser granularity, single-ISA heterogeneous processors attempt to provide support for different granularities of parallelism by integrating multiples types of cores which all use the same ISA [99]. In a similar vein, Kumar et al. discuss the architectural tradeoffs of sharing varying degrees of hardware between processors and threads in a SMT/CMP hybrid design to explore the tradeoffs of ILP and TLP [100].

Coarse-grained reconfigurable architectures: Fisher et al. proposed Custom-fit processors where processor cores are synthesized at design time based on application needs [54]. They adopt a unique approach of designing a heavily customizable VLIW architecture in which the number and types of functional units, memory sizes and hierarchy, and number of registers can all be customized. Through a hardware/software co-design process one important application is taken as input and a customized VLIW architecture heavily optimized for that application is generated. The final processor is fully general purpose and can run all other applications also, albeit not as efficiently as the one "input" application. Tensilica follows a similar approach providing a complete toolchain flow for synthesizing processors and an ISA based on a set of applications [165].

PACT-XPP is an array-based architecture for stream computation which does data-flow computing in the array [19, 58]. Vectorization techniques are used to generate configuration states for this array for large blocks of repetitive code. One of the drawbacks in the architecture is the lack of support for executing sequential programs efficiently and lack of access to random access memory. The Mathstar [69] processor belongs to a new class of chips called Field Programmable Object Array (FPOA), in which, instead of configuration of gates like an FPGA, designers work with a massively parallel array of pre-configured function units like 16-bit ALUs, multiply-accumulate units, and register files which can communicate through an interconnect fabric.

In the ASH architecture, the predication model and dataflow concepts are similar to the TRIPS approach [29]. The main difference being that, ASH targets application-specific hardware for small programs, as opposed to compiling large programs into a sequence of configurations mapped to a programmable substrate. The Garp architecture and the BRASS project used an FPGA based reconfiguration approach to offload compute intensive regions of an application to an on-chip FPGA [76]. Hartenstein has written a literature survey of other reconfigurable coarse-grained architectures targeted at a single application domain [71, 72].

2.2 Data Parallel Architectures

Several authors have proposed architectures and mechanisms for data parallel architectures. In this section we discuss the work most closely related to ours, grouped under vector processors, systolic arrays, SIMD/MIMD processors, stream processing and other hybrid architectures. The key difference between many of these architectures and the polymorphism approach is the ability to support different granularities of parallelism and the granularity of reconfiguration.

Vector processors: Early data parallel architectures were classic vector processors which were built using expensive SRAMs for high-speed memory and large vector register files [138, 112, 78]. These machines were designed for programs with regular control and data behavior, but could tolerate some degree of irregular (but structured) memory accesses using scatter and gather operations. Programs with frequent irregular memory references or accesses to lookup tables performed poorly. A number of architectures have been proposed or built to overcome the limitations of the rigid vector execution model and to allow for dynamic instruction scheduling and conditional execution [49, 94, 48, 149]. Removing these limitations still did not make these architectures widely applicable as they provided support only for a subset of data parallel programs. The Vector IRAM architecture is another vector processing architecture that exploits VLSI density and uses embedded DRAM with closely integrated vector lanes [93]. However, the global control between the different vector lanes and specilization of the vector lanes renders sequential and non-vectorizable code very inefficient on this architecture. Short vector processing has found its way into commercial microprocessors in the form of instruction extensions such as MMX, SSE2, Altivec and VIS [43]. These architectures have similar requirements of regular control and data access, and have further restrictions on data alignment. Some of the ISA extensions, such as MMX and SSE2, have poor support for scalar-vector operations, only operating on one sub-word of a MMX/SSE2 register when using a scalar register as one operand.

Systolic architectures: Systolic arrays were proposed by Kung and Leiserson for processing data in regular fashion in which an array of identical processing elements are interconnected in a pipelined manner, with each element performing the same operation (or operations) and passing along the processed data to its neighbors [81]. Prior to this formal definition and specification of systolic arrays, the British Colossus computer employed an architecture similar to systolic arrays for code breaking during World War II [35]. In general, systolic arrays have primarily been used to build special-purpose application specific hardware [136]. The Warp machine used a systolic array to construct a programmable data parallel architecture to support scientific computing and signal processing applications [10]. The iWarp architecture extended the design of the Warp machine, by designing an iWarp block that could be replicated and connected to form a parallel processor [24]. A single iWarp chip consisted of a processing core and a *communication agent* which orchestrated the communication between different iWarp chips. The iWarp architecture was also targeted at scientific and image processing applications. executing parallel programs on a large iWarp system consisting of many iWarp blocks, using a hybrid multithreading and systolic processing model.

SIMD/MIMD processors: The SIMD and MIMD terms were coined by Flynn in his taxonomy of computer architectures [55]. The early fine-grained SIMD machines like the CM-2 [33] and MasPar MP-1 [21] provided high ALU density but lacked support for fine-grained control and latency tolerance to irregular memory accesses. Modern programmable graphics processors consist of a very wide SIMD execution engine to perform fragment and vertex processing [36]. Several researchers have examined the use of these architectures for more general purpose scientific computation beyond just graphics processing [2]. MIMD architectures have typically been used to build large scale parallel architectures. Other examples include graphics pipelines [5] and video processing [26]. The Briarcliff architecture is a fine-grained MIMD architecture that uses register channels to communicate between independent processing units and by making these channels visible to the compiler allows slack between the independent streams [63]. The use of register channels in this architecture is similar to the uses of FIFOs in the Instruction Level Distributed Processing architecture [90]. The most prevalent use of fine-grained MIMD processing is in modern graphics processors which contain vertex shaders that are MIMD architectures [108, 109].

Stream processors: Stream processing, which has similarities to vector processing and SIMD computation, is being explored in several architectures targeted at multimedia processing. The stream processing paradigm is based on defining a series of compute-intensive operations, also called kernel functions, which consume and produce streams of data, while sequencing through these kernel functions. These kernel functions are in turn applied to each element in the stream. Imagine is a SIMD/vector hybrid using a SIMD control unit coupled with a memory system resembling a vector machine [135]. Other on-chip MIMD architectures such as Merrimac and RAW also target this style of stream processing using sophisticated compiler analysis and programming language techniques [39, 60]. The Brook programming language provides support for stream computation on graphics hardware [28].

Hybrid architectures: Recent proposals have suggested combining vector computation units with modern out-of-order processors. The Tarantula architecture uses a heterogeneous computation approach and integrates a 32 wide vector core and a high performance out-of-order EV8 core to target data-level parallelism and instruction-level parallelism [48]. Tarantula provides a pure vector model of execution with global synchronization between the different vector lanes with partitioned vector registers and optimized accesses to the regular L2 cache for vector loads. The designers went to great lengths to provide the high bandwidth required out of the L2 cache with an innovative conflict-free address generation scheme to maximize the number of concurrent accesses to different cache banks for many types of strided accesses [145]. Pajuelo et al. proposed speculative dynamic vectorization in which vectorizable code segments are detected in sequential code and are speculatively executed on a dedicated vector datapath [122]. This architecture is also heterogeneous since it provides two dedicated datapaths each specialized for a different function.

Intrinsity is an embedded processor that includes a high performance scalar MIPS32 core integrated with an array based parallel vector math unit [121]. The vector math unit consists of an array of ALUs connected to each other using a high bandwidth inter-ALU network fed by a high bandwidth L2 cache. The L2 cache can sustain a bandwidth of 64 Gbytes/sec, when running at 2 Ghz. The instruction control in the array is strict SIMD with each ALU executing the same instruction every cycle. The Cell Broadband Engine(TM) and a trademark of Sony Computer Entertainment, Inc. is another example of a hybrid architecture that includes an in-order processor and up to eight SIMD processors, dubbed Synergistic Processor Engines (SPE), with a software managed memory system [84, 125, 79]. The in-order processor manages memory for the SPEs and is used to program DMA engines that orchestrate DRAM to on-chip memory transfers.

2.3 Scalable Architectures

With transistor counts approaching one billion, tiled architectures are emerging as an approach to manage design complexity. The RAW architecture pioneered research into many of the issues facing tiled architectures such as the complexity of each tile, network interconnect used for communication between the tiles, instruction scheduling across tiles, and efficient memory access across tiles [169, 104, 156, 157, 158]. In the RAW architecture, all tiles are identical and include a processor core, a router, memory ordering logic, and data storage which is configured as a data cache. The Pira$\tilde{n}$ha architecture explored tiled architectures targeted at server workloads and took an extreme position, for the time [18]. It integrated eight very simple cores along with a complete cache hierarchy, memory controllers, coherence hardware, and network controller, all on a single chip built using ASIC 0.18μm technology. Another tiled architecture that uses homogeneous tiles is Smart Memories [107]. The Synchroscalar [120] and AsAP [174] architectures are other examples of homogeneous tiled architecture which are less general and instead specifically targeted at DSP applications. Emerging fine-grained CMP architectures, such as Sun's Niagara [92, 97] or IBM's Cell [84], can also be viewed as tiled architectures. Other examples of tiled architectures targeted at specific domains include Starcore [171], Picochip [66], Clearspeed [67], and Silicon Hive [68], many of which are reviewed in [96].

Each of these architectures implement one or more complete processors per tile. In general, these tiled architectures are interconnected at the memory interfaces, although RAW allows register-based inter-processor communication. TRIPS differs in two ways: (1) different types of tiles are composed to create a uniprocessor and (2) TRIPS uses distributed control network protocols to implement functions that would otherwise be centralized in a conventional architecture.

2.4 Microarchitecture Techniques for ILP

We conclude this literature review by discussing work related to extracting instruction-level parallelism. The dataflow execution model and scalable techniques for extracting ILP are the mostly closely related areas.

Dataflow: The execution model and ISA design for the TRIPS processor is heavily inspired by prior dataflow computers. Dennis and Misunas proposed a static dataflow architecture in their seminal paper on dataflow computing [40]. The amount of concurrency that static dataflow could extract was limited because data tokens could not be produced by an instruction until the tokens produced by it during a previous dynamic instance were consumed. As a result, the levels of concurrency that can achieved by overlapping multiple iterations of a loop is limited. Dynamic dataflow addresses this problem by dynamically labeling dataflow arcs and managing these in a hash table of dataflow tokens [12]. Continuing this work on dynamic dataflow Arvind and Nikhil proposed the MIT Tagged-Token Dataflow architecture with purely data-driven instruction scheduling for programs expressed in a dataflow language [13]. Culler et al. later proposed a hybrid dataflow execution model where programs are partitioned into code blocks made up of instruction sequences, called threads, with dataflow execution between threads [38]. The rich history of dataflow architectures is reviewed by Arvind and Culler [11]. The TRIPS approach differs from these in that we use a conventional programming interface with dataflow execution for a limited window of instructions, and rely on compiler instruction mapping to reduce the complexity of the token matching.

ILP: Processor architectures are driven in equal measure by VLSI technology constraints and performance requirements. Future technology limits of power, design complexity, and wire delays have led architects towards scalable and modular designs. Processor performance in the future, at least in part, must come exploiting more parallelism, and specifically instruction-level parallelism. Extracting ILP creates three requirements for processor architectures: 1) a large window of useful program instructions, 2) a scalable execution core that can examine and execute a large number instructions concurrently, and 3) a high bandwidth and low latency memory system.

Ranganathan and Franklin described an empirical study of decentralized ILP execution models [132]. Sohi et al. proposed Multiscalar processors, in which a single program is broken up into a collection of speculative tasks [150]. A different approach to creating a distributed window uses dynamic traces for the execution partitions [167]. In that work, Vajapeyam and Mitra proposed renaming temporary registers within a trace to reduce the needed global register file and rename bandwidth. More recently, Kim and Smith proposed the ILDP architecture where a distributed microarchitecture using FIFO-based instruction issue queues execute instructions which have been broken into strands of dependent instructions [90].

Other current research efforts targeting ILP are focused on large-window parallelism by means of checkpointing and speculation [37, 152], hybrid dataflow speculation [15], and out-of-order processor frontend microarchitecture mechanisms [119]. In this chapter we have described work that is most relevant to this dissertation. Nagarajan presents a more detailed survey of approaches to ILP in his dissertation [114].

Chapter 3

EDGE ISAs

As a result of technology constraints, RISC and CISC ISAs present significant overheads when extracting concurrency and are becoming increasingly hard to implement. We introduce a new class of ISAs called Explicit Data Graph Execution (EDGE) ISAs which express dependences directly in the ISA and thus enable efficient support for concurrency in the hardware. EDGE architectures provide a technology scalable approach for exploiting concurrency and provide a good starting substrate for developing the concepts of polymorphism to support different granularities of parallelism.

The concept of the EDGE ISA was jointly developed with Ramadass Nagarajan that started with our intial work on Grid Processor Architectures [117]. A more detailed description of EDGE ISAs, its fundamental contributions, compilation strategies for this ISA model, and a detailed performance of the architecture are subjects of his dissertation [114].

In this chapter, we describe EDGE ISAs, how they lend support for polymorphism, and conclude with an overview of the compilation techniques for such ISAs.

3.1 EDGE ISAs

Explicit Data Graph Execution (EDGE) architectures allow compiler-generated dataflow graphs to be mapped to an execution substrate. The two defining features of an EDGE ISA are:

1. Block-atomic execution.
2. Efficient dataflow-like execution enabled by instruction-to-instruction communication within a block. The ISA uses the dataflow graph as the fundamental layer of abstraction to express concurrency to the hardware.

Support for Polymorphism: We use this architectural support of dataflow encoding in the ISA to exploit different granularities of parallelism efficiently. The dataflow encoding is efficient at expressing ILP, TLP, and DLP. This dataflow graph abstraction amortizes the overheads of instruction management across several instruction in a full block of instructions. For extracting ILP, the dataflow encoding expresses the limited parallelism in blocks, small regions of a program, directly to the hardware. The hardware uses control speculation techniques to determine the sequence of blocks and determines the data dependences between blocks through register renaming and load/store dependence checking. For extracting TLP, the dataflow encoding expresses the limited parallelism in each thread, and the hardware can interleave multiple dataflow graphs in the hardware, similar to the SMT approach of interleaving multiple instructions from different thread contexts. For extracting DLP, the dataflow graph abstraction directly expresses the abundant parallelism to the hardware – typically the graphs are very large when programs have data-level parallelism. In conventional RISC and CISC ISAs which require the hardware to rediscover parallelism, the overheads of instruction management affect the scalability of hardware and limit performance. The block atomicity amortizes these overheads across many instructions and expresses dependences efficiently to the hardware.

Technology Scalability: EDGE ISAs amortize per-instruction bookkeeping over a large number of instructions and reduce the accesses to centralized structures thus enabling technology scalability. In particular, the number of branch predictions, number of register file accesses, and complexity of the register renaming hardware is reduced. Furthermore, encoding dependences explicitly in the instructions simplifies dependence checking hardware and obviates the need for hardware to discover parallelism. Finally, EDGE ISAs also reduce the frequency at which control decisions about what to execute must be made (such as fetch or commit), providing latency tolerance to make distributed execution practical. Ranganathan et al. quantify the branch prediction latency tolerance provided by such an architecture [133].

3.2 Execution Model

The execution model for EDGE ISAs treats a block of instructions as an atomic unit for fetching, executing, and committing. The execution substrate is a collection of ALUs, each of which is architecturally visible and named. For simplicity, we assume that all ALUs are homogeneous and can execute any instruction.

Block-atomic execution: In the block-atomic execution model, instructions are placed into blocks by the compiler. Blocks may include predicated instructions but have no internal transfers of control; taken branches (and the last instruction in a block) transfer control to a succeeding block. A block could thus be a basic block, a predicated hyperblock [106], or a run-time trace [137].

Dataflow graph abstraction: The ISA allows the dataflow graph of execution to be directly encoded in the blocks. The data used and consumed by a block are of three types: (1) *block inputs*, which are values produced by preceding blocks and must be read when the execution of the block begins, (2) *block outputs*, which are values created within the block and used by subsequent blocks, and (3) *block temporaries*, which are values that are produced and consumed entirely within the block. Block temporaries can be forwarded directly from producers to consumers, without ever being written back to any central storage. The dataflow graph is encoded in the block through instruction-to-instruction communication of these block temporaries. Block outputs, however, must be written to a central storage like a register file when the block commits. The dependence between block outputs of one block and the block inputs of its successor, along with load-store communication pairs, create the dataflow arcs for the entire program. The output of control transfer instructions which specify the address of the succeeding block are also treated as block outputs. Modifications to memory are maintained in temporary storage until the block is committed.

3.2.1 Block Execution

The compiler statically assigns each instruction in a block to one of the named ALU instruction slots. Each ALU can have multiple instruction slots associated with it. Special `read` instructions, used to read block inputs, are assigned to the register file. Execution of an instruction block proceeds as follows: A block is first fetched and mapped onto the ALUs in the execution substrate at once. Each instruction in the block is stored in the instruction slot at the ALU (similar to a reservation station) to which it was statically assigned. The `read` instructions issued at the register file read block inputs and trigger the dataflow execution by injecting the values to appropriate ALUs.

When all of an instruction's operands have arrived at an ALU, the instruction is executed. This data-driven execution model is similar to that of a traditional dataflow machine [13, 40]. When the instruction completes, its result is forwarded to the ALUs holding consuming instructions, and/or to the register file if the result is a block output.

Operands are delivered directly from producers to consumers (point-to-point) in the ALU network rather than being broadcast to all ALUs. As a result, unlike conventional architectures, which require complex bypass logic between ALUs, a simple point-to-point network will suffice for EDGE architectures. Since all operands are forwarded to the location where instructions are buffered, an instruction does not encode the source locations or register names of its inputs, only its outputs. The physical destinations of the instruction's result are encoded explicitly into an instruction.

When all of the instructions in a block have completed, the block is *committed*. Block outputs are written back to the register file and updates to memory are carried out. Subsequently, the block is removed from the ALUs, and the next block is mapped onto the execution substrate. In the event of an exception being raised by any instruction in a block, the entire block is re-executed after the the exception is serviced. Similar to pipelined execution of instructions for RISC and CISC architectures, implementations of this execution model may overlap both fetch, mapping, and execution of the subsequent block (or blocks) with the execution of the current block. With this type of overlap, multiple blocks can be in flight simultaneously and the ALUs in the execution array can have instructions from many blocks mapped at once, with the dataflow firing rules taking care of the ordering of instructions.

3.2.2 Key Advantages

The block-atomic model will be effective if the number of instructions in the block is large enough to yield long dependence chains that can benefit from the ALU chaining in the execution substrate. The experimental results in Chapter 6 show that compiler-generated block sizes are significant, when predication is used to eliminate control flow hazards.

When we started this research we performed several empirical studies to explore the feasibility of this architecture. Our initial results, published in [140] convinced us of the potential of this architecture and execution model. In that study, we used the Trimaran compiler infrastructure [162] using the SPEC CPU2000 and SPEC CPU95 workloads to measure the properties of blocks that are important for EDGE ISAs: a) the size of blocks, b) number of block inputs, c) number of block outputs, d) number of block temporaries, and e) fanout of block temporaries. Our initial evaluation indicated that programs were well suited for this architecture. Typical block sizes ranged from 27 to 125 dynamically executed instructions, which are sufficiently large to amortize scheduling overheads. The number of input and output values required for a large fraction of the blocks was less than 10 in most of the benchmarks, indicating that the amount of register file communication between blocks is small. The average number of temporary registers per block was larger, ranging from 10 to 30, depending on the benchmark. This range indicates that a substantial amount of communication to the centralized register file can be eliminated through the producer/consumer communication as block temporaries. Finally, the average number of consumers of a produced value is only 1.9, which shows that the network within the execution substrate does not require large bandwidth for intra-block communication.

This execution model addresses several of the challenges for microprocessor performance scaling. In particular, an implementation of this model requires no centralized, associative issue window, no instruction-by-instruction register renaming table and there are fewer register file reads and writes. Despite the lack of these structures, instructions can execute in an order determined at runtime based upon true data dependences, without expensive hazard checking or a broadcasting bypassing and forwarding network. Palacharla et al. demonstrated that broadcast bypass networks scale poorly and typically their complexity grows quadratically with the number of nodes on the network [123]. In other work, we present a taxonomy to classify the entire class of on-chip networks, and propose Routed Inter-ALU networks (RIANs) as a scalable communication network for future processors [143].

The explicit concurrency expressed in the ISA, and static mapping of instructions to resources naturally allows for a scalable and modular microarchitecture implementation. Furthermore, if the physical instruction layout corresponds to the dataflow graph, communication from producers to consumers will take place along short, point-to-point wires. Instructions off of the critical path can afford longer communication latencies between more distant ALUs. The physical layout of ALUs is exposed to the instruction scheduler, so that the wire and communication delays can be used to help the scheduler minimize the critical path. Other publications extensively characterize and analyze this scheduling problem [115, 34, 116].

3.3 Compilation

Architectures work best when the subdivision of labor between the compiler and the microarchitecture matches the strengths and capabilities of each. For future technologies, current execution models strike the wrong balance: RISC relies too little on the compiler, while VLIW relies too much. RISC ISAs require the hardware to discover instruction-level parallelism and data dependences dynamically. While the compiler could convey them, the ISA cannot express them, forcing out-of-order superscalar architectures to waste energy reconstructing that information at run time. VLIW architectures, conversely, put too much of a load on the compiler. They require that the compiler resolve all latencies at compile time to fill instruction issue slots with independent instructions. Since unanticipated runtime latencies cause the machine to block, the compiler's ability to find independent instructions within its scheduling window determines overall performance. Since branch directions, memory aliasing, and cache misses are unknown at compile time, the compiler is unable to generate schedules that best exploits the available parallelism in the face of variable latency instructions such as loads.

EDGE-architectures and their ISAs provide a proper division between the compiler and architecture, matching their responsibilities to their intrinsic capabilities, and making the job of each simpler and more efficient. Rather than packing together independent instructions like a VLIW machine, which is difficult to scale to wider issue, the compiler simply expresses the data dependences through the ISA. The hardware's execution model handles dynamic events like variable memory latencies, conditional branches, and the issue order of instructions, *without* needing to reconstruct any compile-time information.

An EDGE compiler has two new responsibilities in addition to those of a classic optimizing RISC compiler. The first is forming large blocks with no internal control flow for spatial scheduling. The second is the spatial scheduling itself, statically assigning instructions in a block to ALUs in the execution array, with the goal of reducing inter-instruction communication distances and increasing parallelism.

Scale Compiler: In the TRIPS project, the compiler team led by Kathryn McKinley and Doug Burger re-targeted the Scale research compiler [111] to generate optimized TRIPS code. Scale is a compilation framework written in Java that was originally designed for extensibility and high performance on RISC architectures, such as Alpha and Sparc. Scale provides classic scalar optimizations and analysis such as constant propagation, loop invariant code motion, dependence analysis, and higher-level transformations such as inlining, loop unrolling, and interchange. Jim Burrill, Aaron Smith, Bill Yoder, Bert Maher, and Nick Nethercote developed several components to re-target the Scale compiler for TRIPS [146]. To generate high-quality TRIPS binaries, the compiler team added several features to the Scale compiler. Bert Maher and Aaron Smith developed several transformations, including loop transformations and function inlining techniques to generate large predicated hyperblocks [146, 105]. Katherine Coons, Ramadass Nagarajan, Xia Chen, and Sundeep Kushwaha developed the scheduler that maps instructions to ALUs and generates scheduled TRIPS assembly in which every instruction is assigned a location on the execution array [34, 116]. Behnam Robatmili developed the register allocator for the re-targeted compiler. Aaron Smith led the development of predication support in the compiler [147].

Although the past 2 years of compiler development have been labor-intensive, the fact that we were able to design and implement this functionality in Scale with a small development team is a testament to the balance in the architecture; the division of responsibilities between the hardware and the compiler in an EDGE architecture is well suited to the compiler's inherent capabilities. Scale is now able to compile C and FORTRAN benchmarks into full executable TRIPS binaries.

3.4 Summary

The key advantages of EDGE ISAs are higher exposed concurrency and more power-efficient execution. An EDGE ISA provides a richer interface between the compiler and the microarchitecture: The ISA directly expresses the dataflow graph that the compiler generates internally, instead of requiring the hardware to rediscover data dependences dynamically at runtime, an inefficient approach that out-of-order RISC and CISC architectures currently take.

Today's out-of-order issue RISC and CISC designs require many inefficient and power-hungry structures, such as per-instruction register renaming, associative issue window searches, complex dynamic schedulers, high-bandwidth branch predictors, large multiported register files, and complex bypass networks. Because an EDGE architecture conveys the compile-time dependence graph through the ISA, the hardware does not need to rebuild that graph at runtime, eliminating the need for most of those power-hungry structures. In addition, direct instruction communication eliminates the majority of a conventional processor's register writes, replacing them with more energy-efficient delivery directly from producing to consuming instructions.

In this chapter, we described EDGE ISAs and the execution model. In the next chapter, we describe the TRIPS ISA which is one instance of an EDGE ISA and a distributed microrachitecture that implements the ISA. The modular nature of the microarchitecture provides natural support for polymorphism.

Chapter 4

TRIPS Architecture and Prototype Chip

The TRIPS architecture is an instance EDGE ISAs introduced in the previous chapter. The TRIPS microarchitecture is heavily partitioned and uses well defined communication networks to build large, coarse-grained processors (also known as Grid Processors) to achieve high performance on single-threaded applications with high ILP. Unlike conventional large-core designs, which rely on centralized components making them difficult to scale, the TRIPS architecture is heavily partitioned to avoid such structures and long wire runs. These partitioned computation and memory elements are connected by point-to-point communication channels that are exposed to software schedulers for optimization. The processor and memory system is augmented with polymorphous features that enable the compiler or run-time system to subdivide the core for explicitly concurrent applications of different granularities.

The TRIPS architecture is constructed of modular blocks and hence provides a good starting baseline for exploring polymorphism. The key challenge in defining polymorphous features for TRIPS is to balance their appropriate granularity so that workloads involving different levels of ILP, TLP, and DLP can maximize their use of the available resources, and at the same time avoid escalating complexity and non-scalable structures. The TRIPS system employs coarse-grained polymorphous features at the level of memory banks and instruction storage to minimize software complexity, hardware complexity and configuration overheads. In the remainder of this chapter, we describe the TRIPS instruction set, the TRIPS processor microarchitecture, and the prototype TRIPS chip. The following chapter builds upon the architecture description here to present polymorphism and describes the implementation of polymorphism in the TRIPS architecture.

The design and implementation of the TRIPS architecture and the prototype chip has involved many people. Many mechanisms in the architecture like the memory disambiguation, control flow prediction, and on-chip network are subjects of other dissertation. In particular, the core ideas in the processor microarchitecture, the ISA and the execution model were jointly developed by Ramadass Nagarajan and me. The detailed specification and design of the ILP microarchitecture mechanisms including the global control protocols, register renaming mechanisms, tradeoffs in predication strategies, and performance evaluation of the architecture were developed by Ramadass Nagarajan. He was also instrumental in developing our benchmark simulation infrastructure, several hand-optimizations, and detailed analysis of bottlenecks in the microarchitecture and TRIPS ISA. Through the remainder of this chapter, I also indicate the modules in the architecture that were developed by other members of the TRIPS design team.

4.1 The TRIPS ISA

The TRIPS ISA is an example of an EDGE architecture, which aggregates up to 128 instructions into a single block that obeys the block-atomic execution model, meaning that a block is logically fetched, executed, and committed as a single entity. While details of the TRIPS ISA can be found in [110, 142, 147] this section summarizes the most relevant features.

4.1.1 TRIPS Blocks

Each TRIPS block consists of 128 locations, one for each of the possible 128 instructions. The compiler constructs blocks and assigns each instruction to a location. Each block is composed of between two and five 128-byte chunks by the microarchitecture. As shown in Figure 4.1, every block includes a header chunk which encodes up to 32 `read` and up to 32 `write` instructions that access the 128 architectural registers. The read instructions pull values out of the registers and send them to compute instructions in the block, whereas the write instructions return outputs from the block to the specified architectural registers. In the TRIPS microarchitecture, each of the 32 read and write instructions are distributed across the four register banks, as described in the next section.

The header chunk also holds three types of control state for the block: a 32-bit "store mask" that indicates which of the possible 32 memory instructions are stores, block execution flags that indicate the execution mode of the block, and the number of instruction "body" chunks in the block. The store mask is used for distributed detection of block completion.

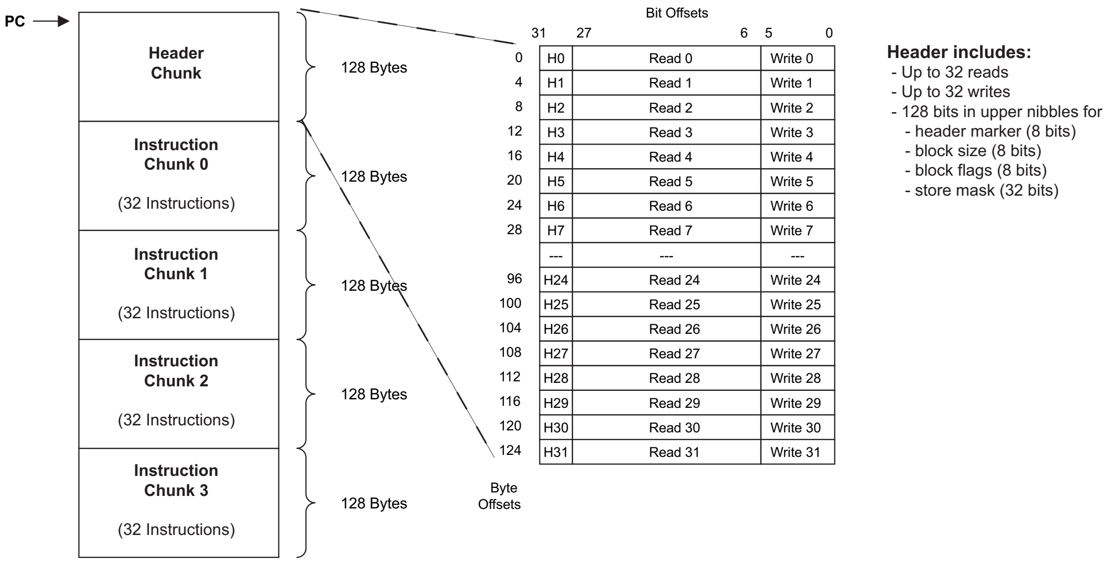


Figure 4.1: TRIPS Block Format.

A block may contain up to four body chunks–each consisting of 32 instructions–for a maximum of 128 instructions, at most 32 of which can be loads and stores. In addition, all possible executions of a given block must always emit the same number outputs (stores, register writes, and one branch) regardless of the predicated path taken through the block. This constraint is necessary to detect block completion on the distributed substrate. The compiler is responsible for generating blocks that conform to these constraints [146].

4.1.2 Direct Instruction-Instruction Communication

Direct instruction communication, in which instructions in a block send their operands directly to consumer instructions within the same block in a dataflow fashion, permits distributed execution by eliminating the need for any intervening shared, centralized structures such as an issue window or a register file between the producer and consumer.

As shown in Figure 4.2, the TRIPS ISA supports direct instruction communication by encoding the consumers of an instruction as targets within the producing instruction, allowing the microarchitecture to determine where the consumer resides and forward a produced operand directly to its target instruction(s). The nine-bit target fields (T0 and T1) shown in the encoding each specify the operand type (left, right, predicate) with two bits and the target instruction with the remaining seven. A microarchitecture supporting this ISA will determine where each of a block's 128 instructions is mapped, thereby determining the distributed flow of operands along the dataflow graph within each block. An instruction's number is implicitly determined by its position in the chunks shown in Figure 4.1.

A second aspect of the instruction encoding is placement. While the 9-bit targets simply create the linkages, the underlying processor microarchitecture is exposed to the compiler so it can generate efficient placement, with the goal of minimizing communication distance among instructions. Nagarajan et al. describe the other aspects of this placement problem and introduce a terminology of classifying architectures based on when (static or dynamic) instruction placement is done and when (static or dynamic) instructions are issued [116]. Burger et al. classify other architectures according to this terminology [31].

Other non-traditional elements of this ISA include the "PR" field, which specifies whether each instruction is predicated on an incoming true or false predicate, and the load/store identifier (LSID) field, which specifies the sequential order

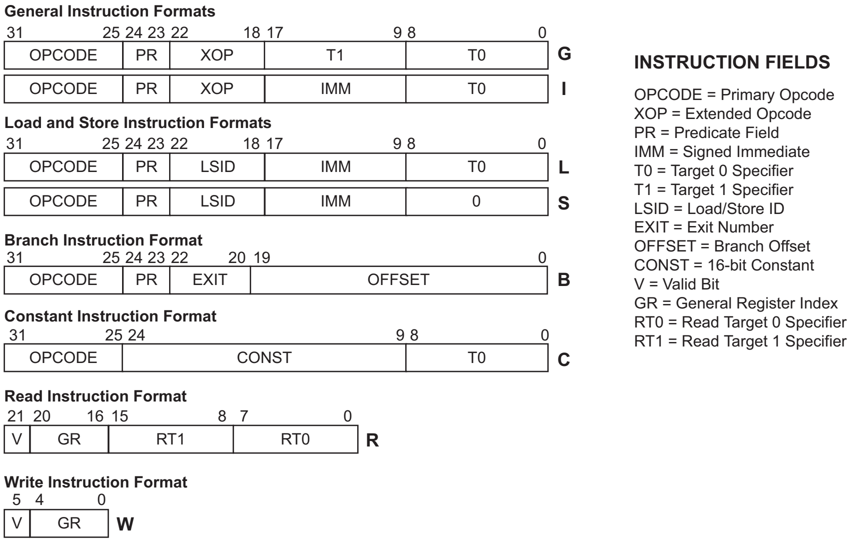


Figure 4.2: TRIPS Instruction Formats.

in which loads and stores must execute. The TRIPS ISA manual contains a complete description of the instruction set architecture [110].

4.2 TRIPS Microarchitecture Principles

The goal of the TRIPS microarchitecture is to achieve high concurrency, whether ILP, TLP, or DLP, on a technology-scalable, distributed core. Our definition of *scalable* and *distributed* is a processor that has no global wires, is built from small set of reused components sitting on routed networks, and can be extended to a wider-issue implementation without recompiling source code or changing the ISA. The three synergistic principles behind this style of microarchitecture are:

Modularity: The microarchitecture is constructed with a small set of tiles replicated and connected together as necessary.

Tiled nature: The microarchitecture is physically partitioned and tiled in nature. The logical organization of the tiles has a physically tiled organization as well. The tiled nature allows a hierarchical design flow at all stages of the design–specification through RTL coding, verification, and physical design. While modularity refers simply to the logical construction of the architecture through a small set of units, tiling refers to a regular spatial placement of module and interconnection among them.

Interconnection networks: The tiles (modules) communicate through well-defined interconnection networks, which in turn have well-defined flow control, *proven* deadlock avoidance, and scalability properties [61].

As a result of the above principles, this microarchitecture is composable, permitting different numbers and topologies of tiles in new implementations with only moderate changes to the tile logic and no changes to the software model.

4.3 TRIPS Microarchitecture Implementation

The TRIPS prototype chip implements an EDGE ISA called the TRIPS ISA. In the following paragraphs we describe the microarchitecture of this prototype chip. Figure 4.3 shows the tile-level block diagram of the TRIPS prototype. The three major components on the chip are two processors and the secondary memory system. The processor cores occupy the top- and bottom-right quadrants of the chip, and the on-chip memory system occupies the left half of the chip. Each processor core is a 16-wide issue TRIPS core that can have up to 1024 instructions in flight.

The secondary memory system includes a set of tiles that are configured to form a NUCA cache [89], two integrated SDRAM controllers, a DMA controller, two chip-to-chip (C2C) controllers that are used to communicate to other TRIPS chips, and an External Bus Controller (EBC) that is used to interface to a PowerPC chip.

The tiles in the processor core and the tiles in the on-chip network are connected internally by one or more micronetworks. We define micronetwork as: *a network that employs many of the traditional networking techniques, such as flow control, but which implements a microarchitecture function that is invisible to software.* In separate work, we describe a taxonomy for classifying these networks based on the physical implementation and the routing protocols used [143]. The taxonomy classifies interconnection networks based on the underlying communication model (broadcast or point-to-point), network architecture (mulit-hop or single-hop), and type of routing control (static or dynamic). Taylor et al. describe another taxonomy for classifying such micronetworks based on a tuple quantifying delays at different points in the network from source to destination [157].

Each of the processor cores is implemented using five unique tiles: one global control tile (GT), 16 execution tiles (ET), four register tiles (RT), four data tiles (DT), and five instruction tiles (IT). The major processor core micronetwork is the operand network (OPN), shown in Figure 4.4. It connects all the tiles except for the ITs in a two-dimensional, wormhole-routed, 5x5 mesh topology. The OPN has separate control and data channels, and can deliver one 64-bit data operand per link per cycle; a control header packet is launched one cycle in advance of the data payload packet to accelerate wakeup and select for bypassed operands that traverse

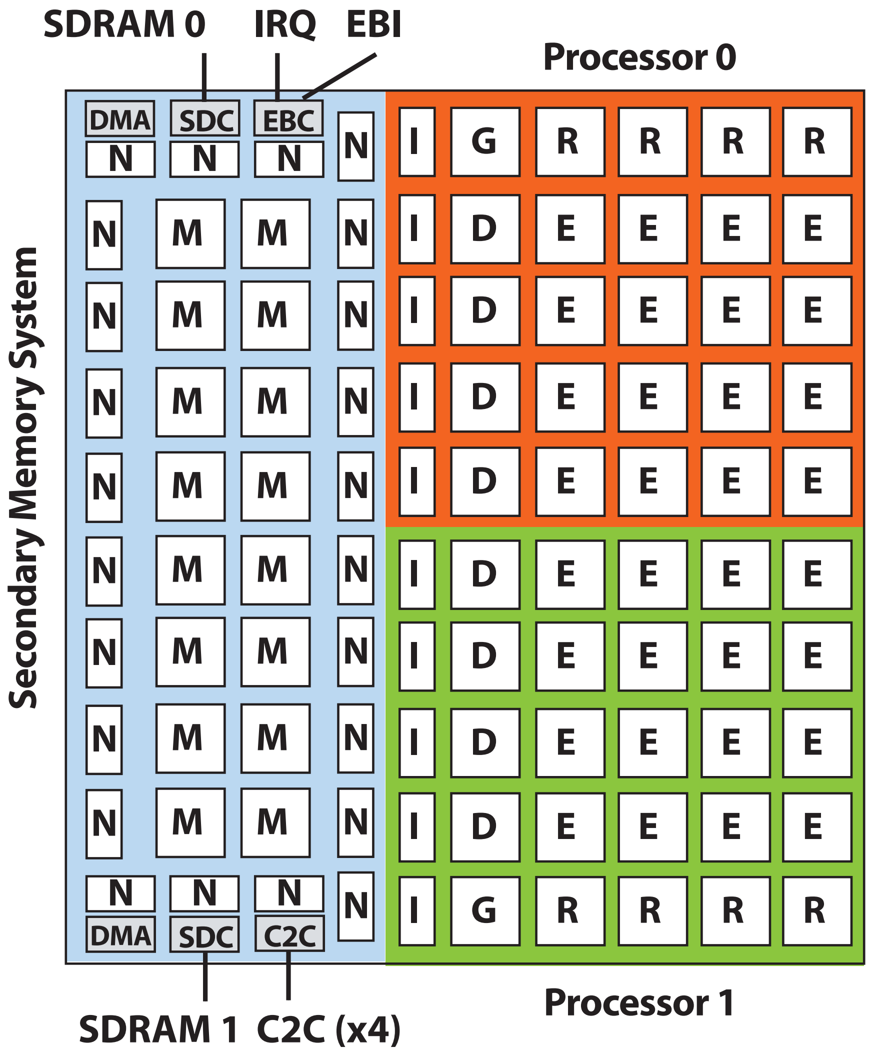
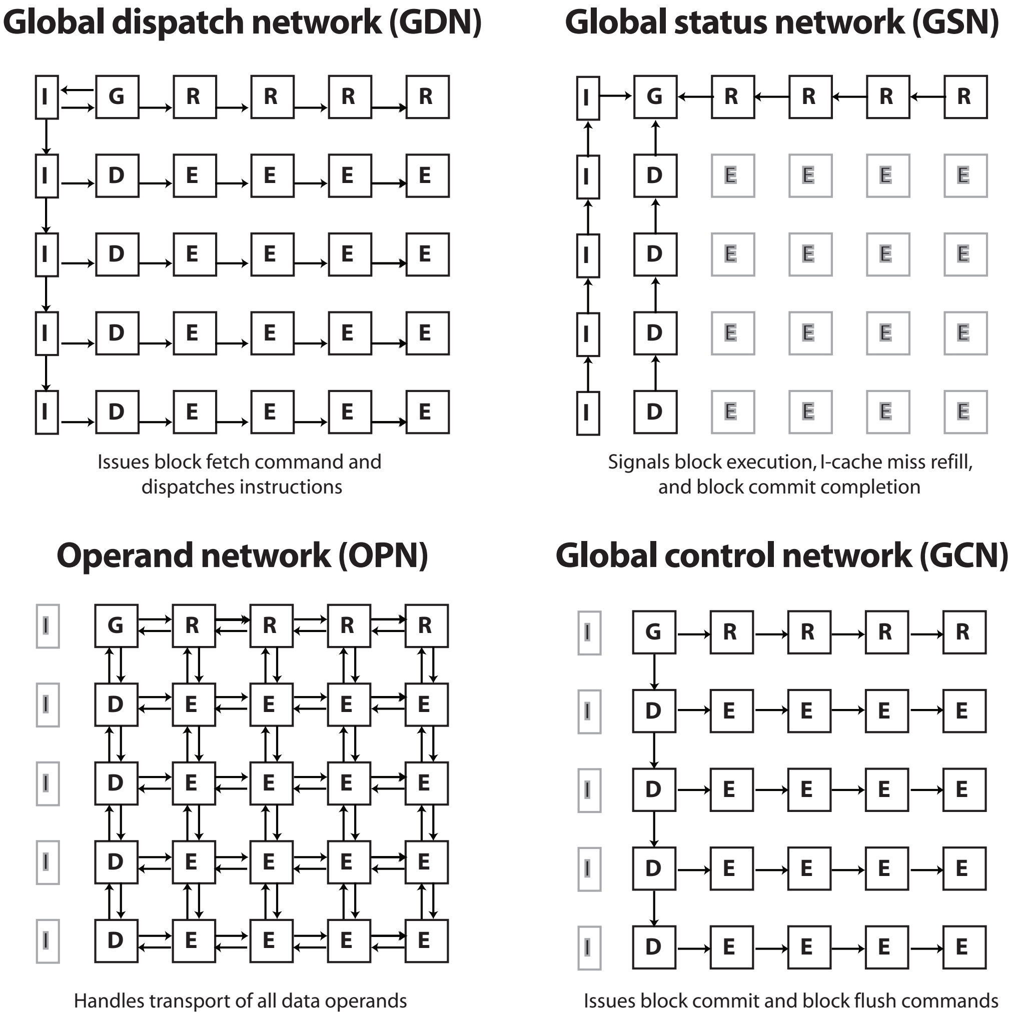


Figure 4.3: TRIPS Prototype Chip Schematic



Global dispatch network (GDN)

Issues block fetch command and dispatches instructions

Global status network (GSN)

Signals block execution, I-cache miss refill, and block commit completion

Operand network (OPN)

Handles transport of all data operands

Global control network (GCN)

Issues block commit and block flush commands

Figure 4.4: TRIPS Micronetworks (GRD, DSN, and ESN not shown).

Micronetwork	Function
Operand network (OPN)	Pass data operands between tiles
Global dispatch network (GDN)	Dispatch instructions to tiles
Global control network (GCN)	Commit and flush blocks
Global status network (GSN)	Transmit information about block completion
Global refill network (GRN)	I-cache miss refills
Data status network (DSN)	Communicate store completion status among the L1 data cache tiles
Extenal store network (ESN)	Determine the completion status of stores in the L2 cache or memory.

Table 4.1: TRIPS processor micronetworks.

the network.

Each processor core contains six other micronetworks as described in Table 4.1. Links in each of these networks connect only nearest neighbor tiles and messages traverse one tile per cycle. We show the links for four of these networks in Figure 4.4 and discuss their usage later in this section.

The particular arrangement of tiles that we implemented in the prototype produces a core with 16-wide out-of-order issue, 64KB of L1 instruction cache, 32KB of L1 data cache, and 4 SMT threads. The microarchitecture supports up to eight TRIPS blocks in flight simultaneously, seven of them speculative if a single thread is running, or two blocks per thread if four threads are running. The eight 128-instruction blocks provide an in-flight window of 1,024 instructions.

The two processors on the chip have independent micronetworks. To communicate, they must go through the secondary memory system, in which the On-Chip Network (OCN) is embedded. The OCN is a 4x10, wormhole-routed mesh

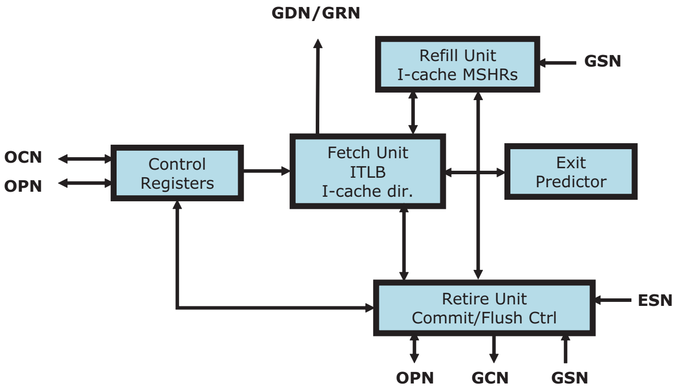


Figure 4.5: TRIPS Tile-level Diagrams: Global Tile - GT

network, with 16-byte data links and four virtual channels. The network is optimized for cache-line sized transfers (one header packet followed by four 16-byte data packets), although other request sizes are supported for operations like loads and stores to uncacheable pages. The OCN acts as the transport fabric for all inter-processor, L2 cache, DRAM, I/O, and DMA traffic.

In the rest of this section, we describe the contents of each processor core tile, and then in Section 4.4, show how global operations among the tiles–such as flush and commit–are implemented by distributed microarchitectural protocols.

4.3.1 Global Control Tile (GT)

The GT is the only singleton tile in the processor. As shown in Figure 4.5, it holds the block program counter (PC) and handles all TRIPS block management: prediction, fetch, dispatch, completion detection, flush (on mispredictions and interrupts) and commit. It also holds the control registers that configure the processor into different speculation, execution, and threading modes. Thus, the GT inter-

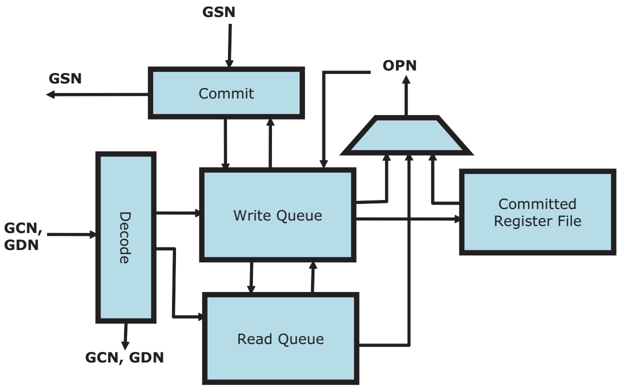


Figure 4.6: TRIPS Tile-level Diagrams: Register Tile - RT

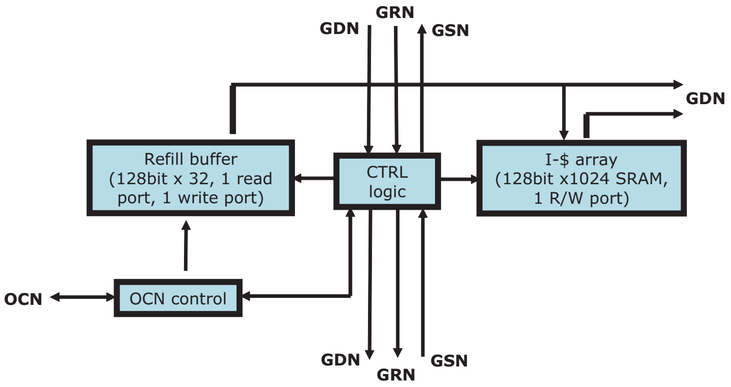

GDN
GRN
GSN
GDN
Refill buffer (128bit x 32, 1 read port, 1 write port)
CTRL logic
I-$ array (128bit x1024 SRAM, 1 R/W port)
OCN
OCN control
GDN
GRN
GSN

Figure 4.7: TRIPS Tile-level Diagrams: Instruction Tile - IT

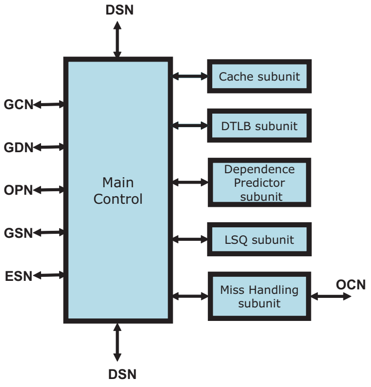


Figure 4.8: TRIPS Tile-level Diagrams: Data Tile - DT

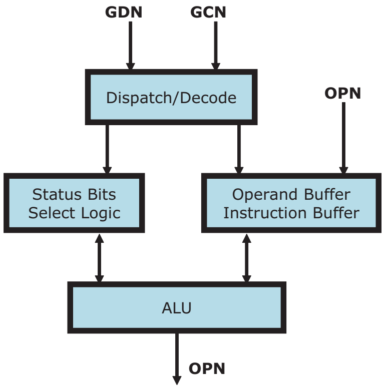


Figure 4.9: TRIPS Tile-level Diagrams: Execution Tile - ET

acts with all of the control networks, as well as the OPN for reading and writing the block PC. The major structures in the GT are the instruction cache tag arrays, the instruction TLB, and the next-block predictor. Ramadass Nagarajan was the primary designer of the global tile logic and Nitya Ranganathan and Ramadass Nagarajan jointly developed the next-block predictor.

The GT maintains the state of all eight in-flight blocks. When at least one of the block slots are free, the GT accesses the block predictor, which takes three cycles and emits the predicted target address of the next block. Each block may emit only one “exit” branch, even though it may contain several predicated branches. The block predictor uses a branch instruction’s three-bit exit field to construct exit histories instead of using taken/not-taken bits. The predictor has two major parts: an exit predictor and a target predictor. The predictor uses those exit histories to predict the next three-bit block exit, employing a tournament local/gshare predictor similar to the Alpha 21264 [88] with 9K, 16K, and 12K bits in the local, global, and tournament exit predictors, respectively.

When the exit number is predicted, it is combined with the predicting block address to access the target predictor to predict the next-block address. The target predictor contains four major structures: a branch target buffer (20K bits), a call target buffer (6K bits), a return address stack (7K bits) and a branch type predictor (12K bits). The BTB predicts targets for branches, the CTB for calls and the RAS for returns. The branch type predictor predicts the type of the branch currently being predicted (call/return/branch/sequential-branch). The type predictor is necessary because of the architecture's distributed fetch protocol; the predictor never sees the actual branch instructions, since they are sent directly from the ITs to the ETs, so the branch type must be predicted.

4.3.2 Instruction Tile (IT)

The ITs simply act as slave I-cache banks for the GT, which holds their tags. As shown in Figure 4.7, each IT contains a 2-way, 16KB bank of the L1 I-cache. Since each TRIPS block consumes as many as 640 bytes worth of instructions, the microarchitecture breaks blocks into five 128-instruction chunks, caching each chunk in one respective IT. Each 16KB IT bank can thus hold a 128-byte chunk for each of 128 blocks. The Instruction Tile was designed and implemented in Verilog by Haiming Liu.

4.3.3 Register Tile (RT)

Centralized register files cause power and delay problems in large, out-of-order processors. The TRIPS microarchitecture partitions its register file into banks, with one bank in each RT. Like the other tiles, register banks are nodes on the OPN, allowing the compiler to place instructions that read and write from/to a given bank close to that bank if they appear critical. The RT was designed and implemented in Verilog by the author along with Steve Keckler.

Since many def-use pairs of instructions are converted to intra-block temporaries by the compiler, and thus never access the register file, the total register bandwidth requirements are reduced by approximately 70%, on average, compared to a RISC or CISC processor. The four distributed banks can thus provide sufficient register bandwidth with a small number of ports; in the TRIPS prototype, each RT bank has two read ports and one write port. Since the TRIPS ISA specifies 128 architectural registers, each of the four RTs contains one 32-register bank for each of the four SMT threads that the core supports for a total of 128 registers per RT.

In addition to the four per-thread architectural register file banks, each RT contains two other major structures: a read queue and a write queue, as shown in Figure 4.6. These queues contain the eight read and eight write instructions from the block header for each of the eight blocks in flight, and are used to forward register writes dynamically to subsequent blocks reading from those registers. The read and write queues perform an equivalent function to register renaming for a physical register file in a superscalar processor, but were less complex to implement due to the ISA support for read and write instructions.

4.3.4 Execution Tile (ET)

As shown in Figure 4.9, each of the 16 ETs consists of a fairly standard single-issue pipeline, a bank of 64 reservation stations, an integer unit, and a floating-point unit. The ET design team was led by Premkishore Shivakumar and included Nitya Ranganathan and Divya Gulati who developed the verification infrastructure for this tile. All units are fully pipelined except for the integer divide unit, which takes 24 cycles. The 64 reservation stations hold eight instructions for each of the eight in-flight TRIPS blocks. Each reservation station has fields for two 64-bit operands data operands and a one-bit predicate.

4.3.5 Data Tile (DT)

The four DTs, each of which is a client on the OPN, each hold one 2-way, 8KB bank of the 32KB L1 data cache, as shown in Figure 4.8. The DTs design was led by Simha Sethumadhavan and Robert McDonald developed the verification infrastructure for this tile. Virtual addresses are interleaved across the D-tiles at the granularity of the D-tile's 64B cache-line. In addition to the L1 cache bank, each DT contains a copy of the load/store queue (LSQ), a dependence predictor, a one-entry back-side coalescing write buffer, a data TLB, and a MSHR that can support up to 16 requests for up to four outstanding cache lines.

Because the DTs are distributed in the network, we implemented a *memory-side* dependence predictor, closely coupled with each data cache bank. Loads issue from the ETs, and a dependence prediction occurs in parallel with the cache access only when the load arrives at the DT. The dependence predictor in each DT uses a

1024-entry bit vector. When an aggressively issued load causes a dependence misprediction (and subsequent pipeline flush), the dependence predictor bit to which the load address hashes is set. Any load whose predictor entry contains a set bit is stalled until all prior stores have completed. Since there is no way to clear individual bit vector entries in this scheme, the hardware clears the dependence predictor after every 10,000 blocks of execution.

The hardest challenge in designing a distributed data cache was the memory disambiguation hardware. The TRIPS ISA restricts each block to 32 maximum issued loads and stores. Since eight blocks can be in flight at once, up to 256 memory operations may be in flight. However, the mapping of memory operations to DTs is unknown until their effective addresses are computed. The two resultant problems are (a) determining how to distribute the LSQ among the DTs, and (b) determining when all earlier stores have completed–across all DTs–so that a held-back load can issue.

We solved the LSQ distribution problem largely by brute force. Centralizing the LSQ would have resulted in poor performance and too much complexity, as loads would have to be routed to two places and then synchronize on the appropriate action. Partitioning the LSQ among the DTs was problematic since we had no low-overhead solution for handling overflow of one of the partitions. Instead, we replicated four copies of a 256-entries LSQ, one at each DT. This solution is unscalable and wasteful (since the maximum occupancy of all LSQs is 25%), but was the least complex alternative for the prototype. The LSQ can accept one load or store per cycle, forwarding data from earlier stores as necessary. If there is a partial in-flight match (e.g. multiple store byte instructions feeding a single, later load word instruction), the load consumes one cycle for each store that forwards a piece of the load.

4.3.6 Secondary Memory System

The TRIPS prototype supports a 1MB static NUCA [89] array, organized into 16 Memory Tiles (MTs), each one of which holds a 4-way, 64KB bank. Each MT also includes an on-chip network (OCN) router and a single-entry MSHR. Each bank may be configured as an L2 cache bank or as a scratch-pad memory, by sending a configuration command across the OCN to a given MT. By aligning the OCN with the DTs, each IT/DT pair has its own private port into the secondary memory system, supporting high bandwidth into the cores for streaming applications. The Network Tiles (NTs) surrounding the memory system act as translation agents for determining where to route memory system requests. Each of them contains a programmable routing table that determines the destination of each memory system request. By adjusting the mapping functions within the TLBs and the network interface tiles (N-tiles), a programmer can configure the memory system in a variety of ways including as a single 1MB shared level-2 cache, as two independent 512KB level-2 caches (one per processor), as a 1MB on-chip physical memory (no level-2 cache), or many combinations in between. We refer the reader to [89] for more details on the cache organization, and [61] for details on the TRIPS On-Chip Network. The other six tiles on a chip's OCN are I/O clients, namely two SDRAM controllers, two DMA controllers, one Chip-to-Chip controller, and one external bus controller that can interface to a PowerPC440GP chip, which acts as a host processor. Paul Gratz and Changkyu Kim designed and implemented the M-Tiles, N-tiles, the C2C controller and the SDRAM controllers, and Saurabh Drolia, Sibi Govindan, and Simha Sethumadhavan implemented the other controllers.

4.4 Microarchitecture Execution Model

As defined by the ISA, block execution is atomic, and the main challenge is to support this logical view of atomic block execution with speculative execution on a physically distributed microarchitecture occurring under the covers. To execute a block in this microarchitecture, the following four logical steps must be performed:

1. Fetch: fetch instructions from memory
2. Execution: the actual execution of the individual instructions in the block
3. Completion: detect that all the instructions in a block that must execute have completed execution. Since blocks can have predicated instructions, not all the instructions in a block need to actually execute during every dynamic invocation of a block.
4. Commit: update architecture state modified by a block.

Additional steps are required when an exception is detected in a block and these steps are carried out instead of commit. Since the processor core is physically distributed, different parts of the block are fetched from different tiles, execution happens in a distributed fashion across the different tiles, and the architecture state itself is stored across different tiles. Table 4.2 summarizes the timeline of block execution and shows how the different micronets interact to create the logical view of atomic block execution.

Below we illustrate with a detailed example, the execution of block instructions alone, leaving out the fetch, complete, and commit steps. A detailed description of timing diagrams and the implementation of the microarchitecture pipeline can be found in [142]. Figure 4.10 shows an example of how a code sequence is executed on the RTs, ETs, and DTs. Figure 4.11 shows the encoding for a single instruction and how the microarchitecture interprets the instruction bits to map instructions to reservation stations in an ET. All of the operands described are delivered over the OPN. The code starts when the read instruction R[0] is issued to RT0. It reads the value either from architectural register R4 or from the write queue of a prior in-flight block that writes to R4. That value is sent to the left operand of two instructions, the `teq` (N[1]) and the `muli` (N[2]).

When the test instruction receives the register value and the immediate "0" value from the `movi` instruction, it fires and produces a predicate which is routed to the predicate field of N[2]. Since N[2] is predicated on false (indicated by the `p_f` prefix), if the routed operand has a value of 0, the `muli` will fire; if the predicate's value is 1, N[2] will not issue. If it issues, N[2] multiplies the arriving left operand by four, and sends the result to the address field of the `lw` (load word). Note that if N[2] does not fire due to a mismatched predicate, the dependent load will not fire, as it will never receive its left operand.

Event	**Micronet**	**Tiles**	**Description**
		Fetch	
Refill	GRN	GT, IT	Check if block exists in cache, if not send commands to ITs to fetch block from secondary memory system into the cache
Dispatch	GDN	GT, IT, ET, RT, DT	Send instructions from instruction cache banks to different tiles
		Execute	
Execute	OPN	ET, RT, DT, GT	Instructions execute in data flow fashion within the block
	DSN	DT	DTs use the DSN network for memory disambiguation
		Completion or Exception	
Completion	GSN	RT, DT, GT	RTs and DTs send a complete command to the GT when all reads and stores have been received at the RTs and DTs respectively
Exception	GSN	RT, DT, GT	If exception detected on a memory access or read, information is passed on to the GT
		Commmit or Flush	
Commit	GCN	RT, DT, GT	GT sends a commit command to RTs and DTs; architecture state updated
Flush	GCN	RT, DT, GT	GT sends a flush command to RTs and DTs in case of exception or misspeculation; temporary buffers cleared, internal state machines are reset
Commit-ack	GSN	RT, DT, GT	RTs and DTs send acknowledge command when architecture state completely update. This two-phase commit, commit-acknowledge creates the logical view of atomic block commit

Table 4.2: Block execution timeline and micronets used.

```
R[0]       read R4     N[1,L] N[2,L]
N[0]       movi #0     N[1]
N[1]       teq         N[2,p] N[3,p]
N[2] p_f   muli #4     N[32,L]
N[3] p_t   null        N[34,L] N[34,R]
N[32]      lw #8       N[33,L]             LSID=0
N[33]      mov         N[34,L] N[34,R]
N[34]      sw #0                           LSID=1
N[35]      callo       $foo
```

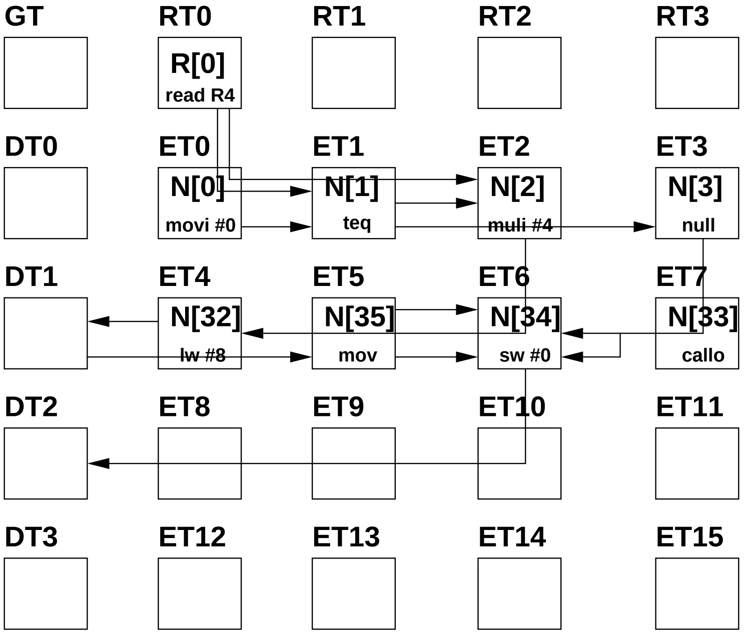


Figure 4.10: TRIPS execution example.

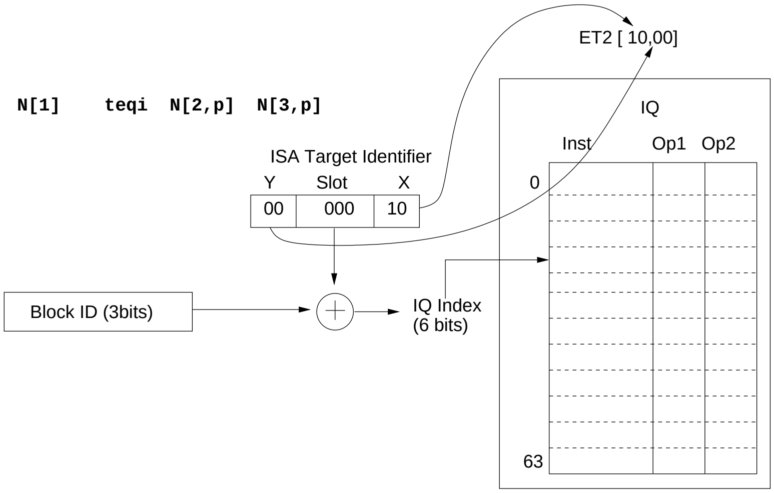


Figure 4.11: Encoding of a single instruction and mapping instructions to reservation stations.

If the load fires, it sends a request to the pertinent DT, which responds with the value of the load and routes it to N[33]. The DT uses the load/store IDs (0 for the load and 1 for the store, in this example) to ensure that they execute in the proper program order if they share the same address. The result of the load is fanned out by the `mov` instruction to the address and data fields of the store. If the test predicate is true (indicated by `p_t`), however, the `null` instruction instead fires, also targeting the address and data fields of the `sw` (store word). Note that although two instructions are targeting each operand of the store, only one of those instructions will fire due to the predicate. When the store is sent to the pertinent DT and the block-ending call instruction is routed to the GT, the block has produced all of its outputs and is ready to commit. Note that if the store is nullified, it does not affect memory, but simply signals the DT that the store has issued. Nullified register writes and stores are used to ensure that the block always produces the same number of outputs for completion detection.

4.5 TRIPS Prototype Chip

The physical design and implementation of the TRIPS chip were driven by the principles of partitioning and replication. The physical design and floorplan directly represents the logical organization of TRIPS tiles connected only by point-to-point, nearest-neighbor networks. The microarchitecture principles of modularity, tiling, and communication through well defined networks, are directly reflected in the physical design and simplified the physical design process.

The only exceptions to our nearest neighbor communication restriction are the global reset signal, the “processor halted” signal from the GTs to the external bus controller (EBC), and the “processor halt” command from the EBC to the GTs. All of these signals are latency tolerant, however, and all are pipelined heavily across the chip.

Hierarchical design has been common practice for quite some time. Example include system-on-a-chip (SOC) designs that aggregate components with different functions via a portable communication network or bus, and chip-multiprocessor (CMP) designs, in which a processor can be replicated many times on the chip. TRIPS differs from SOCs and CMPs in that the individual tiles are designed to have diverse functions but cooperate together to implement a more powerful and design-scalable uniprocessor. In the following two sub-sections, we first provide a detailed specification of the TRIPS chip and then briefly discuss the physical design aspects of the chip.

4.5.1 Chip Specifications

The TRIPS chip is implemented in the IBM CU-11 ASIC process, which has a drawn feature size of 130nm and 7 layers of metal. The chip itself includes more than 170 million transistors in a chip area of 18.30mm by 18.37mm, which is placed in a 47.5mm square ball-grid array package. The TRIPS chip design team included faculty, staff, and graduate students at UT-Austin and an IBM Microelectronics ASIC design team located in Austin, TX. UT-Austin was responsible for all architecture, logic design, verification, and timing. IBM supplied the physical design methodology and libraries, and was responsible for the physical design tasks including test infrastructure insertion, the final physical floorplan, placing and routing of all cells, and the tapeout process.

The final clock period at worst case process parameters is 4.5ns, which accounts for pessimistic clock skew and wiring parasitics from the final layout. To first order, this corresponds to approximately 32 fanouts of 4 (where 1 FO4 is the latency for a single inverter to drive four copies of itself). By comparison, leading edge custom microprocessors are in the range of 15-20 FO4 [4]. A custom design style coupled with a more experienced design team, some amount of re-pipelining and more time devoted to timing optimization would likely be able to drive the TRIPS architecture into that same regime. Adding a more aggressive process and less conservative gates than a standard ASIC process would make the TRIPS clock rate competitive with that of a high-end commercial microprocessor.

Figure 4.12 shows an annotated floorplan diagram of the TRIPS chip taken directly from the design database as well as a coarse area breakdown by function.

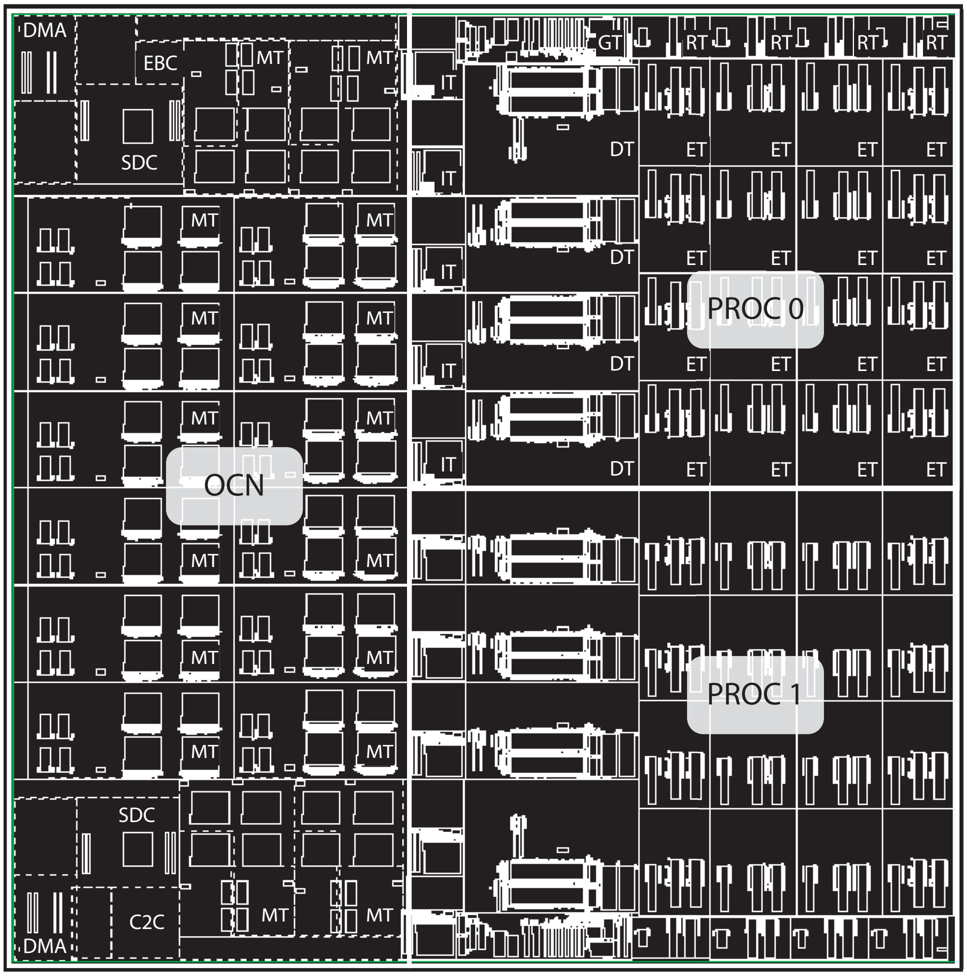


Figure 4.12: Floorplan diagram

Overall Chip Area	
29%	Processor 0
29%	Processor 1
21%	Level-2 Cache
14%	On-chip Network
7%	Other (controllers, etc.)

Processor Area	
30%	Functional Units (ALUs)
4%	Register Files and Queues
10%	Level-1 Caches (I and D)
13%	Instruction Queues
13%	Load/Store Queues
12%	Operand Network
2%	Next block predictor
16%	Other

Table 4.3: Chip area breakdown

The diagram shows the boundaries of the TRIPS tiles, as well as the placement of register and SRAM arrays within each tile. We did not label the network tiles (NTs) that surround the OCN since they are so small. Also, for ease of viewing, we have omitted the individual logic cells from this plot. Table 4.3 lists the area breakdown of the major components of the chip. Each instance of a tile was individually placed and routed because IO cells are distributed through the chip and create blockages at different locations in different tiles. As a result all the instances of a tile do not look identical in this floorplan diagram.

Controllers: In addition to the core tiles, the TRIPS chip also includes six controllers that are attached to the rest of the system via the on-chip network (OCN). The two 133/266MHz DDR SDRAM controllers (SDC) each connect to an individual 1GB SDRAM DIMM. The chip-to-chip controller (C2C) extends the on-chip network to a four-port mesh router that gluelessly connects to other TRIPS chips. These links nominally run at one-half the core processor clock and up to 266MHz. Each TRIPS prototype board includes 4 TRIPS chips and ports to extend the system to up to 32 TRIPS chips on 8 boards. The two direct memory access (DMA) controllers can be programmed to transfer data to and from any two regions of the physical address space including addresses mapped to other TRIPS processors; the global physical address map contains memory regions for each processor in the system.

Finally, the external bus controller (EBC) is the interface to an on-board PowerPC control processor. To reduce design complexity, we chose to off-load much of the operating system and runtime control to this PowerPC processor. The EBC allows the PowerPC to read and write all TRIPS chip architectural state (memory, registers, etc.) and relays interrupt requests from TRIPS processors and DMA controllers to the PowerPC, which proxies system calls for the TRIPS chips on the board.

IOs and Test: The TRIPS chip includes nearly 600 signal I/Os, including 108 for each SDRAM interface, 312 for the chip-to-chip controller (39 pins per channel × four directions × input/output per direction), and 69 pins for the EBC. Not shown in Figure 4.12 are the individual I/O cells, which are placed near the periphery of the chip. Some of ETs, MTs, and DTs are larger than others to accommodate the placement of these I/O cells.

Finally, the ASIC methodology requires LSSD scan support for manufacturing testing and JTAG I/O boundary scan. In addition, we and our IBM partners added a scan controller to enable the scan chains to be used for silicon debug in functional mode by allowing scan access to most of the internal state. The TRIPS chip also includes two phase-locked loops (PLLs) to generate the clocks for the four on-chip clock domains (main clock, C2C clock, and two clocks for the DDR SDRAM controller). These clocks are asynchronous to one another and we use synchronizers when crossing the main clock, C2C clock and SDRAM clock boundaries. The C2C interface to other TRIPS chips is clocked in a source-synchronous fashion and incoming C2C packets are synchronized into the local domain before being used.

4.5.2 Physical Design

The TRIPS design flow relies extensively on tile-level partitioning as well as a modular ASIC design flow. As a part of their ASIC services, IBM provides register and SRAM array generators that we used heavily not only for registers and memory, but also for branch prediction tables, instruction queues, and reservation stations. Through a university license, Synopsys provided their DesignWare suite which included synthesizable integer units, floating-point units, queues, and FIFOs. The design-time advantages of the ASIC flow are offset by greater area and slower clock rates relative to a custom design. However, the advantages of tile-level partitioning would apply directly to a custom VLSI design of TRIPS.

Table 4.4 shows additional details on the design of each TRIPS tile. The

Cell Instance column shows the number of placeable instances in each tile, which provides a relative estimate of the logic complexity of the tile. A placeable instance is a pre-defined macro available in the IBM library provided, examples of which include simple 2-input AND gates to SRAMs and register files. *Array Bits* indicates the total number of bits found in dense register and SRAM arrays on a per-tile basis, while *Size* shows the area of a representative of each type of tile. Although the logic for every instance of a tile is identical, each tile was individually placed and routed because IO cells are distributed through the chip and create blockages at different locations in different tiles. The representative area shows the area of one instance for each tile type. *Tile Instances* shows the total number of copies of that tile across the entire chip, and *% Chip Area* indicates the fraction of the total chip area occupied by that type of tile.

As shown in Table 4.4, the DT is certainly the most complex of the tiles, due in large part to the demands of an out-of-order memory system rather than the distributed nature of the TRIPS processor. Its cell count and area is skewed somewhat by the CAM arrays for the maximum sized load/store queues which had to be implemented from discrete latches, because no suitable dense array structure was available. We saw the same phenomenon in OPN and OCN routers. The large cell counts in the ET are due largely to the computational units, such as the floating point units, which are synthesized to the standard cell library rather than implemented using a custom datapath.

Tile	Function	Cell Instances	Array Bits	Size (mm^2)	Tile Instances	% Chip Area
GT	Processor control	51,684	93K	3.1	2	1.8
RT	Register file	26,284	14K	1.2	8	2.9
IT	Instruction cache	5,449	135K	1.0	10	2.6
DT	L1 Data cache	119,106	89K	8.8	8	21.0
ET	Instruction execution	83,887	13K	2.9	32	28.0
MT	L2 Data cache	60,115	542K	6.5	16	30.7
NT	OCN NW interface and routing	23,467	–	1.0	24	7.1
SDC	DDR SDRAM controller	64,441	6K	5.8	2	3.4
DMA	DMA controller	30,365	4K	1.3	2	0.8
EBC	External bus controller	28,547	–	1.0	1	0.3
C2C	Chip-to-chip communication controller	47,714	–	2.2	1	0.7
	Totals (for entire chip)	5.8M	11.5M	334	106	100.0

Table 4.4: TRIPS Tile Specifications.

4.5.3 Design Analysis

Verification The partitioned nature of the TRIPS chip facilitated a highly hierarchical verification strategy. Each of the 11 tile design teams created a sophisticated self-checking testbench for their tile that employed both directed and random tests to exercise as many of the corner cases as possible. The random tests varied both test inputs and the timing of responses to tile requests. To assess coverage, we augmented each tile design with event counters, and ensured that the counters were exercised, all lines of Verilog were hit, and that the internal state machines hit all of the pertinent states. The tile design approach also provided opportunity for concurrent development and verification of the tiles before putting the tiles together and verification of the processor core or the full chip.

We also spent four person-months on performance verification. Using a suite of microbenchmarks, with some randomly generated programs, we reduced the average error between the low-level performance simulator and the RTL simulator from 10% on average to 3%. This effort uncovered sixteen performance bugs, ten of which turned out to be worth the effort to fix. The three most significant ones were fixing the issue priority in the ET, reducing the flush penalty by one cycle, and reordering predictor operations to eliminate an occasional pipeline bubble before issuing a fetch.

4.6 My Contributions

In this section, I briefly summarize my specific contributions to the implementation of the prototype chip, which was done using a design team of more than 10 people. Ramadass Nagarajan, Robert McDonald, Doug Burger, Steve Keckler, and I jointly defined the TRIPS ISA. Along with Ramadass Nagarajan, I co-led the development of our performance simulator, called *tsim_proc* that we used to fine-tune the microarchitecture before embarking on the logic design. During the microarchitecture specification, logic design, and Verilog implementation, my contributions were: implementation of the Register Tile in Verilog, joint specification of the Execution Tile microarchitecture, specification of the operand network, and verification of the OPN.

I led the processor level verification effort which included developing a sophisticated random program generator that we used for verifying the TRIPS implementation at the processor level. I also developed a separate floating point verification suite based on the Softfloat suite [75] to test the floating point implementation in the TRIPS design.

I led the physical design of the chip and my contributions were the chip floorplan and coordinating the individual tile-level floorplans. I also implemented the IO cell assignment for the TRIPS chip which included developing several scripts to analyze routing paths on-board and reduce crossings. Finally, I also analyzed the chip pin signals from an electrical standpoint to determine the maximum noise and delays on the different groups of signals to ensure signal integrity and correctness.

4.7 Discussion

In this chapter, we described the TRIPS ISA which is one instance of an EDGE architecture, its microarchitecture design, and outlined the implementation of the TRIPS prototype chip. The dataflow graph abstraction in the ISA and the scalable, partitioned, modular nature of the microarchitecture provide natural support for polymorphism. The microarchitecture principles of modularity, tiling, and communication through well defined networks, are directly reflected in the physical design and simplified the physical design process. As a result of a hierachical design approach and the highly modular nature of the design, there was significant productivity gains as many of the modules were concurrently developed and verified before being intergrated. The number of unique modules that make up this design is also quite small–only eleven. The prototype chip is a proof of concept for distributed microarchitectures that provide high levels of concurrency.

The prototype chip provides limited polymorphism support, namely explicit thread-level parallelism by sub-dividing the instruction window, reconfiguration of memory banks to provide programmer controlled scratch-pad support, and DMA controllers for orchestrating off-chip to on-chip memory transfers. In the following chapter, we develop the principles of polymorphism and explain the mechanisms in the context of the TRIPS processor architecture. We evaluate the polymorphism mechanisms that are implemented in the TRIPS prototype chip and use a high level simulator to evaluate other polymorphism mechanisms that are not implemented in the prototype.

Chapter 5

Polymorphism in the TRIPS Architecture

Emerging applications with heterogeneous computation needs and future technology constraints have created the need for a design methodology that can achieve economies of scale, provide support for heterogeneous applications, combat processor complexity, and address wire-delay limitations and power. Architectural polymorphism achieves this by altering the behavior of coarse-grained components to support different granularities of parallelism on a programmable architecture. Polymorphism also requires an underlying architecture that can scale with technology and is built using modular microarchitecture blocks. In the previous chapter, we described the TRIPS architecture which provides such a scalable and modular processing substrate. In this chapter, we use TRIPS as the baseline architecture for developing the mechanisms for polymorphism.

The need for architectural mechanisms for distinct application domains has been evident for many years and has in fact been available for almost a decade in a modest fashion in general purpose processors. Multimedia extensions such as Intel MMX/SSE [124], PowerPC Altivec [44], SPARC VIS [161], PA-RISC MAX2 [103], MIPS MDMX [77], and Alpha MVI [1] provide general purpose architectures with a means to exploit small scale data-level parallelism. All of the instruction set extensions coupled with their microarchitecture implementations provide a nascent form of polymorphism. The front-end of the processor is configured slightly differently to read from a separate physical register file, whereas the execution units and some other parts of the internal microarchitecture behave the same way. Typically memory disambiguation hardware and caching operate differently. Simultaneous multithreading (SMT) is a second form of polymorphism which is growing in prevalence in single processor chips and chip multiprocessors [164]. In an SMT processor, the register files, instruction fetch logic, and instruction retirement logic, operate slightly differently, while the execution core of the microarchitecture operates the same whether executing one thread or multiple threads. The register files are replicated to provide separate storage for each thread, the instruction fetch logic is modified to fetch from multiple threads, and the instruction retirement logic is modified to handle speculation for each thread separately.

While this limited polymorphism has been sufficient thus far, future application trends point to a growth in the inherent heterogeneity of applications. Examples include the following:

- **Multimedia databases:** The amount of multimedia data is growing rapidly and different types of computation, like database search and multimedia processing, are required on these databases [45].

- **Games:** The physics computation [23, 98], graphics computation [108], and simulation [23] in games all have different computation needs, with growing computation requirements for all three. Physics computation resembles scientific computation workloads, graphics computation has similarities to scientific computing workloads but typically has many more irregular memory accesses, and game simulation utilizes many recursive data structures operating on many data objects in an irregular fashion with little opportunity for pre-computation of memory addresses before their use.

- **Consumer electronics:** Many consumer electronic devices like cellphones and handheld game devices are expected to perform multiple functions. The OMAP3 architecture is a specification for cellphones and integrates up to six processors, each being dedicated to a separate function including general purpose processing, audio/video decoding and playback, 2D and 3D graphics processing, and peripheral I/O controllers [17]. Several handheld manufactures expect a multitude of processing tasks on a single device: wired (Ethernet), wireless (Wi-Fi), and cellular (3G) communication, storage management, biometric identification, security and digital rights management, 3D sound field, and 3D video processing to name a few [16].

Designing such multiple specific solutions introduces a processor complexity problem. Architectural polymorphism solves this application heterogeneity problem and addresses technology constraints in a complexity-effective manner. We defined polymorphism in chapter 1 as "the ability to modify the functionality of coarse-grained microarchitecture blocks, by changing control logic but leaving datapath and storage elements largely unmodified, to build a programmable architecture that can be specialized on an application-by-application basis." We use complexity-effective in the same sense as Moore's definition of complexity effective processor design [113]:

> A complexity-effective design is a design that: 1) embraces a relatively small set of overriding design principles and associated mechanisms, and 2) has been ruthless in collapsing unnecessary complexity into these more fundamental and elegant mechanisms.

In the remainder of this chapter, we describe in detail the principles of polymorphism, the resources and mechanisms required to implement polymorphism, and explain why these mechanisms are fundamental building blocks for polymorphism.

The TRIPS architecture is used as one specific architecture and microarchitecture to implement and evaluate these mechanisms. Choosing a specific ISA and microarchitecture is necessary for quantitative evaluation. This ISA and microarchitecture are also inherently suited to support polymorphism. The dataflow graph abstraction in the TRIPS ISA directly lends itself to polymorphism as it serves as the unifying abstraction level to express different granularities of concurrency. The distributed and modular nature of the microarchitecture already provides the coarse-grained building blocks that are required for architectural polymorphism.

The principles of polymorphism are not dependent on the TRIPS ISA or microarchitecture. The specific implementation of the mechanisms are tied to TRIPS processor microarchitecture, but the basic mechanisms could be applied to any architecture.

5.1 Principles of Polymorphism

Adaptivity across granularities of parallelism: Polymorphism is intended to provide heterogeneous computation capability and adapt to changing application behavior and demands. As described in Chapter 1, we identify the differences in granularities of parallelism as the fundamental architectural difference between applications. Based on granularities of parallelism, programs can be broken down into three categories: instruction-level parallelism, thread-level parallelism, and data-level parallelism. A polymorphous architecture must be able to adapt to these three granularities of parallelism.

Economy of mechanisms: To be complexity-effective, the polymorphous mechanisms must be few in number and they should provide a set of primitive reconfigurable functionality to microarchitecture blocks that can be used to specialize an architecture on an application-by-application basis, instead of a being a set of fixed function extensions. As a short case study, consider an application that has straight-forward data-level parallelism and operates on two long arrays. The fixed function extension approach would entail building a vector core and interfacing it to a conventional processor and compiling programs into vector instructions. The polymorphism approach, on the other hand, would entail creating mechanisms to modify the instruction fetch, select, and execution logic to provide instruction efficiency and modifying the memory system to provide support for regular memory accesses. These mechanisms are by definition uncoupled, meaning the memory system support can be used in isolation without enabling any of the execution core mechanisms. The design challenge is to determine a small set of mechanisms that give “universal” coverage. Our approach to determining these mechanisms was to identify the basic properties of programs and how they affect the microarchitecture. Based on this analysis, we determine a fundamental set of mechanisms that specialized the microarchitecture on application by application basic.

Granularity of configuration: A polymorphous architecture alters behavior of *coarse-grained* microarchitecture modules, by changing the control logic but reusing datapath and storage elements. Providing application specialization by configuring *fine-grained* blocks can be a challenge. Reconfigurable architectures perform fine-grained reconfiguration to synthesize blocks with different functionality to provide application-by-application specialization of hardware. They have all mostly provided application specific hardware and not programmable hardware. As reviewed in chapters 1 and 2, examples include FPGAs, Tensilica, Pact-XPP, MathStar, Piperench, and ASH. All of these designs work well for a small domain of problems where the application can be easily mapped to the hardware, typically “regular” applications, but perform poorly on general purpose programs. By integrating an FPGA to a conventional processor pipeline, the Garp architecture performs fine-grained configuration on this hybrid programmable substrate [76]. The Garp approach however, targets loop-level parallelism only.

Configuring coarse logic blocks with a small set of mechanisms is better at adapting to different types of programs from a performance perspective. This chapter describes the mechanisms which create a configurable execution core, configurable control flow, and a configurable memory system. In this chapter, we qualitatively justify this approach in terms of design complexity. In the next three chapters we discuss the quantitative performance results that such an approach provides, and in chapter 9 in the conclusions of this dissertation, we provide a broader discussion comparing polymorphism to other approaches.

5.2 Resources

We classify the types of resources in polymorphous architecture into three categories based on their function. In the next section we classify different processor resources into these categories and describe the configuration mechanisms.

Fixed resources: Some resources in the processor operate in the same way regardless of the executing application. For example, the instruction cache always tries to capture as much of a program's instructions as possible and provides low-latency access to the program's instruction stream. Fixed resources are fundamental to the basic operation of the processor and their function remains the same for all types of applications.

Polymorphous resources: The configurable resources in the processor perform different types of operations or change their operation policies, depending on program behavior. For example, instruction fetch logic either fetches from one single program thread all the time, or uses a round-robin scheduling policy to fetch from multiple instruction streams if the processor is configured to execute multiple threads simultaneously.

Specialized resources: Some resources in the processor are specialized for specific functions and may not be utilized at all times, with some applications never needing such functionality. The replicated register file storage in an SMT processor is an example of such a resource. In an SMT processor which supports up to four simultaneous threads, there are four copies of the architectural register file. When only one thread is executing on the processor, three of the register files are completely unused. To be efficient, these application specific resources should be minimized.

The *specialized resources* and *polymorphous resources* provide polymorphous architectures the capability of adapting to application needs. Homogeneous and heterogeneous systems can be analyzed in terms of this resource classification. Heterogeneous systems have only fixed resources and specialized resources - for example the vector register file in the Tarantula architecture is a specialized resource, whereas the execution core is a fixed resource. The Cell processor's SPEs can be considered specialized resources since they are primarily used to execute single precision SIMD code whose data has already been brought into neighboring memory banks [154]. Today's multicore chips and the XBox360 [9] can be viewed as homogeneous systems with only fixed resources providing a single execution model to all programs.

5.3 Mechanisms

The TRIPS ISA expresses concurrency to the hardware by breaking programs into blocks and encoding instruction dependences within these blocks by making the dataflow graph explicit in the ISA. This dataflow graph abstraction is used as the unifying theme across different granularities of parallelism and the mechanisms are built around this dataflow execution model. Below we describe the polymorphous mechanisms with respect to the three main processor components: the execution core, instruction fetch and control, and data storage – both memory and registers.

5.3.1 Execution Core

The TRIPS ISA breaks programs into blocks and encodes dataflow graphs in these blocks. The execution core provides a set of reservation stations on to which these dataflow graphs can be dynamically mapped. These reservation stations, also referred to as block slots (since blocks are mapped to them), form one polymorphous resource and are managed differently based on the application.

Across different granularities of parallelism, the nature of these dataflow graphs can vary, and the types of communication between these dataflow graphs can change as well.

ILP: With sequential codes, where ILP is the dominant type of parallelism, the size of the graphs is quite small – of the order of 20 to 40 instructions. To extract ILP efficiently, the reservation stations are used to map a number of speculatively fetched dataflow graphs, since these graphs are typically small and many such graphs are needed to fill the reservation station space.

TLP: When executing multiple programs, dataflow graphs from different programs must be managed in the execution core to extract TLP. The reservation stations are partitioned across programs and dataflow graphs from multiple programs are mapped to the reservation stations.

DLP: When there is ample data-level parallelism, these graphs can be very large. To extract DLP, since the graphs are large and control flow is regular, the reservation stations are used to hold one single large graph that can be statically generated at compile time.

5.3.2 Control Flow

Depending on the type of parallelism, the control behavior of applications vary quite dramatically. Three control flow mechanisms capture all of the diverse behavior exhibited: 1) Control speculation for ILP, 2) Instruction fetch across threads for TLP, and 3) Optimized instruction fetch to exploit repetitive control flow for DLP. For programs with mostly instruction-level parallelism, it is crucial to have highly accurate control flow prediction, since the control flow is very irregular and is hard to determine statically at compile time. With thread-level parallelism, to optimize the performance across threads, the instruction flow management between threads is an important question to address and introduces policy decisions in building the instruction fetch modules. With programs dominated by data-level parallelism, the control flow behavior is very repetitive and easily predictable. Using control flow speculation techniques can unnecessarily place instruction fetch on the critical path to execution. Instead, we design an optimized instruction fetch mechanism that reuses fetch instructions.

These control flow techniques are not mutually exclusive. Efficiency can be further increased by using limited amount of control speculation within each thread while executing multiple threads. Some programs with DLP are best supported by a fine-grained MIMD substrate and the control flow mechanisms to configure the processor like a MIMD machine are similar to TLP control flow management.

5.3.3 Data Storage

Based on liveness, the duration between definition and last use, data values in programs can be classified as short-term, long-term, and persistent. Short-term data is data whose liveness in a program is within a few lines of code, and in the TRIPS compiler such data are live only within a block or dataflow graph. Long-term data is data whose liveness is typically within a function, and in TRIPS such data are live across blocks. Persistent data is data whose liveness spans several functions and is live for a large fraction of the program's execution. Typically, persistent data is written to memory. In a RISC architecture short-term and long-term values are stored in registers, and persistent data in memory. Polymorphism provides the opportunity to manage these values differently in the hardware based on application needs.

Short-term data: Dataflow graphs are directly mapped to reservation stations and short-term data are data operands passed between nodes in the dataflow graph. These are mapped to reservation stations and the ISA explicitly assigns these values to specific reservation stations.

Long-term data: Long-term data are values passed between dataflow graphs that the compiler has placed in different blocks. These are mapped to the architecture register storage and depending on granularity of parallelism, the register space can be managed differently. When executing only one thread, the physical register space implemented can be used for speculative blocks, and while executing multiple threads, the physical register space is partitioned among multiple threads.

Persistent data (Memory): Programming models used in conventional languages like C, C++, and Java have a simple view of memory used for storing persistent data, with the hardware and the operating system responsible for caching policies and paging. This strategy works well for irregular programs where dynamic behavior is best exploited by observing run-time behavior using hardware. However, when the program behavior is regular and well structured, there is benefit to explicitly managing memory through software. In the TRIPS chip, the on-chip memory is constructed using a tile of interconnected memory banks. These memory banks are exposed to software and can be can be configured to behave as NUCA style L2 cache banks [89], scratchpad memory, or synchronization buffers for producer/consumer communication. In addition, the memory tiles closest to each processor can be augmented with a high-bandwidth interface that enhances access to persistent storage. The Cell and Imagine are other processors that provide explicit memory management. The Streaming Register File architecture of Imagine [135] inspired our design of configuration of L2 storage as scratchpad memories.

5.3.4 Summary

Table 5.1 summarizes these mechanisms and resources involved in implementing these mechanisms. In the following sections we describe the implementation of these mechanisms in the TRIPS architecture. We discuss the mechanisms for ILP, TLP, and DLP in that order.

Parallelism	Resources	Policies
Execution core management		
ILP	Reservations stations	Map multiple dataflow graphs
TLP	Reservation stations	Map multiple dataflow graphs from different threads
TLP	Instruction select logic	Prioritize between threads
DLP	Reservation stations	Map large unrolled dataflow graphs
Data storage management		
ILP	Register files	Register renaming across blocks
TLP	Register files	Storage for architecture state from many threads
DLP	Register files	High register file bandwidth
DLP	Memory system	High bandwidth and software controlled memory management
Control flow management		
ILP	Instruction fetch	Control speculation
TLP	Instruction fetch	Control speculation and fetch multiple threads
DLP	Instruction fetch	Optimize regular control flow - reuse fetched instructions
DLP	Instruction fetch, reservation stations, and instruction select logic	Decoupled sequencing support at each ET creating a MIMD execution model

Table 5.1: Summary of polymorphism mechanisms.

5.4 Instruction-Level Parallelism

In this section, we describe how polymorphism can be used to run single-threaded codes efficiently by exploiting instruction-level parallelism. Previous publications have referred to some of these techniques by referring to them as the D-morph mode of the processor [141].

The primary requirements for achieving high ILP are a large instruction window and resources to exploit concurrency in the instruction stream. To exploit ILP in the TRIPS processor, the reservation stations in the core are configured as a large, distributed, instruction issue window. The direct target encoding in the TRIPS ISA enables out-of-order execution while avoiding the associative issue window lookups of conventional machines. To use the instruction buffers effectively as a large window, the processor must provide high-bandwidth instruction fetching, aggressive control and data speculation, and a high-bandwidth, low-latency memory system that preserves sequential memory semantics across a window of thousands of instructions. In the subsequent sections we describe the implementation of the mechanisms for exploiting ILP.

5.4.1 Execution Core Management

The polymorphous resources in the execution core are the reservation stations that provide instruction and operand storage space. To extract ILP, these reservation stations are configured to behave like an instruction window. Such a configuration uses the reservation stations at each Execution Tile to map dataflow graphs directly to the ETs. This physically distributed issue window spread across the ETs allows orders of magnitude increases in window sizes compared to conventional superscalar processor designs–in the TRIPS implementation we achieve one order of magnitude increase. Since there are multiple reservation stations at each ET and multiple ETs, this window is fundamentally a three-dimensional scheduling region. The x- and y-dimensions correspond to the physical dimensions of the ET array and the z-dimension corresponds to multiple instruction slots at each ET, as shown in Figure 5.1.

To fill one of these 3-D scheduling regions, the compiler schedules blocks by assigning each instruction to one node in the 3-D space. Several policies can be implemented to map the instructions in the ISA to these hardware slots provided by the microarchitecture. In the TRIPS prototype we assume fixed size blocks, and break the instruction window into groups of 128, with each such group being assigned one block of instructions. Recall that with 64 reservation stations at each tile and a total of 16 execution tiles the total instruction window size is 1024.

Figure 5.1a shows a four-instruction block (H0) mapped into the first group of reservation stations. Figure 5.1b shows the detailed mapping of instructions to reservation stations in a group. All communication within the block is determined by the compiler which assigns instructions to reservation stations and with operands dynamically routed directly from ET to ET. Consumers are encoded as an explicit 7-bit target field. The microarchitecture interprets these 7-bits as X, Y, and Z-relative offsets to route operands to targets.

The number of bits that can be specified in the target field implicitly limits the size of the dataflow graphs that the compiler can construct, and hence the size

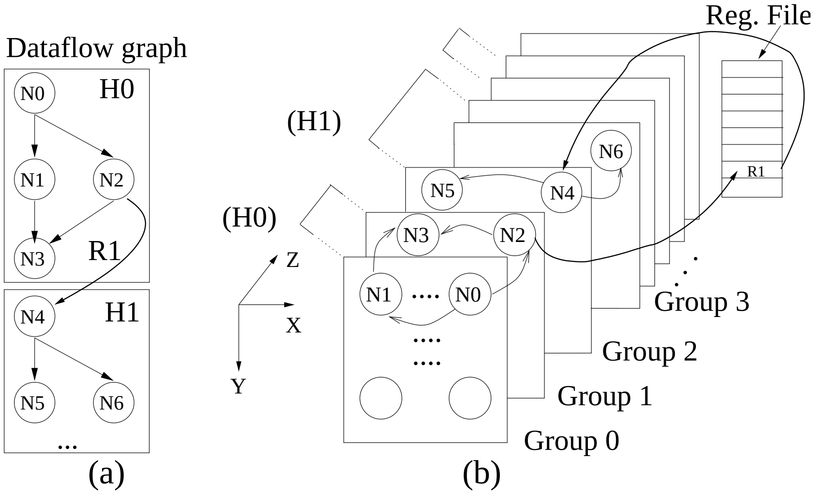


Figure 5.1: Execution core management for ILP.

of the blocks. The number of bits in the target field also directly corresponds to the amount of state the microarchitecture needs to support. Larger graphs can be constructed with a large target field, allowing hard to predict branches to be predicated, thus hiding control flow inside these graphs. The two main challenges in supporting a large target field are the hardware challenge in managing the large amount of state in the microarchitecture and the software challenge in building large dataflow graphs where the number of unused instructions at runtime is small. For the TRIPS prototype chip we chose a 7-bit target field since our experimental results showed block sizes were mostly between 20 and 60 instructions and we expect a block size of 128 to allow us to push the compiler to its limits and explore the design space.

5.4.2 Control Flow Management

To enable an effective large instruction window the processor's control flow logic employs two mechanisms: control speculation to build large instruction windows and high bandwidth instruction fetch.

Control speculation: The compiler is able to generate blocks comprised of dataflow graphs that are between 20 and 60 instructions on average. However, to extract ILP, a much larger window of instructions must be examined and this is achieved by speculating on control flow between blocks. The basic mechanism of providing support for control speculation is two-fold. First, we build a next-block predictor that can predict the next-block to be fetched and executed, similar to a branch predictor used in conventional processor. Second, we manage the reservation stations in the execution core like a circular buffer and map multiple blocks to the instruction window and execute instructions across these block simultaneously. The next-block predictor is a *specialized resource* and the reservation stations form a polymorphous resource, both of whose functions are described below.

Next-block predictor: The next-block prediction is made using a scaled-up tournament exit predictor [82], which predicts a binary value indicating the branch that is predicted to be the exit of the block–recall each block can have multiple branches, of which only one can be taken at runtime. The value generated by the exit predictor is used to index into a set of Branch Target Buffers (BTB) to obtain the next predicted block address. The branch type is also predicted by the exit predictor, and is used to select an address from the multiple BTBs. Ranganathan et al. describe the predictor in further detail [133]. This predictor organization exploits the restriction that each block emits one and only one branch thus avoiding the need to scan the instructions to make the prediction, which permits the predictor to be decoupled from the instruction fetch engine. The per-block accuracy of the exit predictor ranges from 74% to 99%.

Reservation stations: In the TRIPS processor, the total instruction window size provided by the hardware is 1024, with 64 slots available at each of the 16 ETs ($16 * 64 = 1024$). These 64 slots at each ET, are broken into groups of 8. Combining a group of 8 slots across all the ETs provides 128 slots which corresponds to the size of blocks the TRIPS ISA allows: the TRIPS ISA allows only fixed size blocks, with each block containing 128 instructions (unused instructions are encoded as NOPs by the compiler). To map one block of 128 instructions, one group of 8 slots at each ET is combined together ($8 * 16 = 128$). The remaining seven groups are used to map speculative blocks. These groups are managed like a circular buffer with the non-speculative block successively being mapped to group 0, 1, 2, and so on.

High-bandwidth instruction fetch: To fill the large distributed instruction window, the processor includes high-bandwidth instruction fetch mechanisms through the use of a set of partitioned instruction caches. These banks which are in the Instruction Tile (IT) are a fixed resource, meaning that their behavior is the same independent of the type of parallelism. These cache banks are interleaved such that each bank holds 32 of the 128 instructions in a block, and the 32 instructions in each bank correspond to instructions that have been assigned to ETs in the same row as that IT. When there are free reservation stations to map instructions, the control logic accesses a partitioned instruction cache by broadcasting the index of the block to all banks. Each bank then fetches four instructions, one for each ET in a row, with a single access and streams the instructions to the bank's respective row.

5.4.3 Data Storage Management

Short-term data: To extract high ILP, the short-term data operands are mapped to the reservation stations. The management of these short-term data operands forms another fixed resource in the processor. Short-term data operands are operands used in intra-block communication and at the hardware level, this communication maps to operands passed between reservation stations.

Long-term data: Operands are passed between dataflow graphs (or blocks) through registers and their life time in the program spans multiple dataflow graphs. Register renaming in conventional processors creates links between dependent instructions in the instruction window. Similarly, when extracting ILP by speculatively executing dataflow graphs in an EDGE architecture, we must create links between dataflow graphs dynamically, so that the start of execution of a dataflow graph does not have wait until its predecessor has completed and determined to be non-speculative. To manage these long-term data operands efficiently, the microarchitecture implements block-level register renaming to allow rapid passing of values between dataflow graphs, without having to wait for each block's values to be transferred to architecture state.

Persistent data: To support high ILP, the processor memory system must provide a high-bandwidth, low-latency data cache, and must maintain sequential memory semantics to support conventional programming models. The physically distributed data storage in the processor core, comprised of Data Tiles (DT), is configured to behave like a first-level data cache, and the on-chip memory is configured to behave like a second-level data cache. To provide support for ILP, the DTs also include a few specialized resources: 1) MSHRs which track the state of outstanding cache misses, 2) LSQs which detect load/store dependences and enforce the correct ordering of loads and stores in the program, and 3) store merging logic which reduces the number of writes to the cache lines by merging multiple sub-word accesses to the same word in the cache.

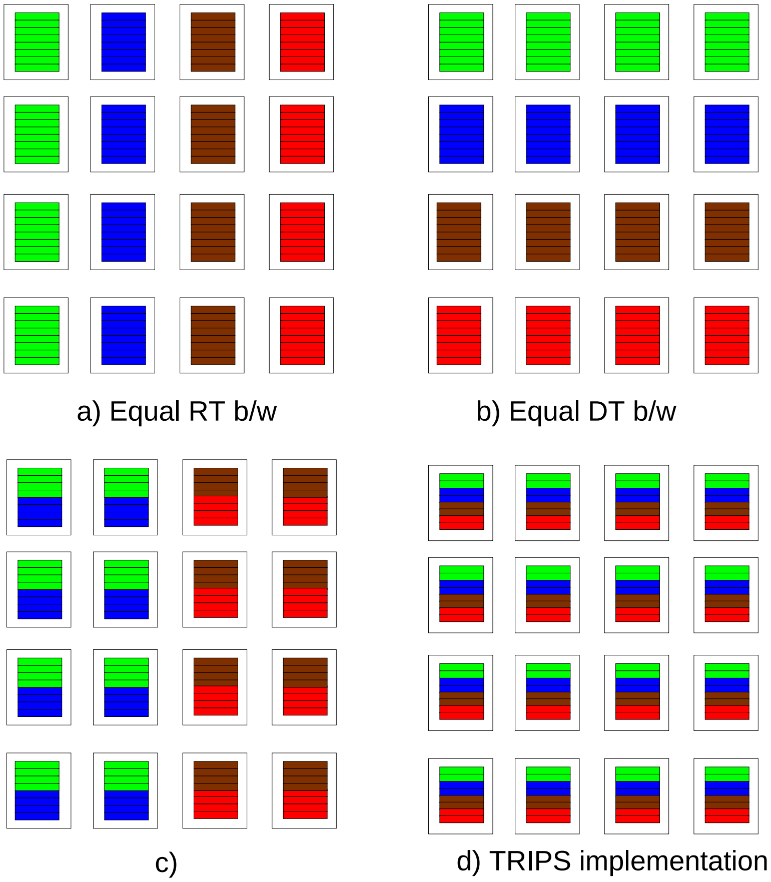
The on-chip memory is configured as a non-uniform cache access (NUCA) array [89], in which elements of a set are spread across multiple secondary banks. The banks have miss-handling logic, a set of tag arrays, and status bits to behave like a cache. The on-chip network also provides a high-bandwidth link to each L1 bank for parallel L1 miss processing and fills. According to the terminology introduced by Kim et al., the TRIPS chip implements a S-NUCA cache.

To summarize, the fixed resources, namely the data caches and instruction caches, the specialized resources, namely, the next-block predictor, MSHRs, LSQs, and store merging logic, and the polymorphous resources, namely the reservation stations configured as an out-of-order issue window and the register renaming logic configured to stitch speculative dataflow graphs together, provide a highly effective distributed processing substrate for extracting ILP.

5.5 Thread-Level Parallelism

When executing applications with thread-level parallelism, high processor utilization can be achieved by mapping multiple threads of control on to a single processor. Tullsen et al. introduced the terminology of simultaneous multithreading (SMT) to refer to fine-grained interleaving of instructions from multiple threads in a processor's pipeline [164]. Previous proposals and implementations of SMT have focused on extensions and modifications to a baseline out-of-order superscalar microarchitecture. In this dissertation, we present a set of polymorphous mechanisms that provide SMT support. By largely sharing datapath and storage elements, our implementation of SMT eliminates some of the replicated structures of previous implementations like multiple reorder buffers.

The basic principle for supporting thread-level parallelism is to split the processor storage resources between multiple threads, and augment the control logic to dynamically share datapath components, like the functional units, between threads. We break the processor storage resources into slices with each slice being assigned to a different thread of control. The control logic is augmented to implement a fairness policy to allow each thread of control to access the datapath. And finally, the architecturally visible storage, namely the register files, are replicated. Within



a) Equal RT b/w

b) Equal DT b/w

c)

d) TRIPS implementation

Figure 5.2: Partitioning execution core resources to support thread-level parallelism. Each color denotes a different thread.

each thread, the processor still extracts ILP, but as each slice is narrower than when running a single program, the ILP extracted per thread is lower. In the following subsections, we discuss the mechanisms that implement SMT through polymorphism.

5.5.1 Execution Core Management

Instead of holding non-speculative and speculative blocks for a single thread as in the case of extracting ILP only, the reservation stations are partitioned *a priori* and assigned to multiple programs (threads). The instruction selection logic in the ETs is augmented to implement a round-robin fair selection scheme between the threads that have a ready instruction to execute. The partitioning of these resources raises two questions:

When to partition: Static partitioning is straight-forward and easy to implement, but can leave processor resources poorly utilized when different threads have different user assigned priorities. While dynamic partitioning can be aware of such application needs, it increases both the hardware and software complexity. Expressing user priorities and policies to the hardware introduces software complexity and dynamic partitioning of processor resources introduces hardware complexity. Hardware profiling based approaches can implement dynamic partitioning without any changes to software.

How to partition: The reservation stations form a 3-D instruction space which can be sliced in different ways to map multiple threads. Figure 5.2 shows a spectrum of partitioning strategies. The main differences between the partitioning schemes are implementation complexity, skewed distance from the register files across threads, skewed distance from the data tiles across the threads, skewed instruction fetch bandwidth and latency. The partitioning strategies shown in (a), (b), and (c) in figure 5.2, add complexity to the instruction fetch logic as the natural alignment of 32 instructions per bank must be changed, or the instruction fetch network must be augmented to route instructions across rows. Figure 5.2d shows the strategy adopted in TRIPS which leaves most of the design unchanged and requires modifications only to the instruction selection logic in the core. Since the TRIPS ISA has fixed 128-instruction blocks, any kind of partitioning strategy must provide at least 128 slots for each thread, and any additional slots can be used for speculation within a thread.

To keep the microarchitecture's execution as close to the ILP model as possible, and to reduce implementation complexity, in the TRIPS prototype chip we implemented a simple sharing scheme denoted in Figure 5.2d. Each thread gets $1/4th$ of the resources, irrespective of how many threads are executing concurrently, and up to 4 threads can be executing simultaneously. The significant drawback of this simplifying decision is that when only two threads are executing, half of the processor's reservation station are unused.

5.5.2 Control Flow Management

Control flow management mechanisms to support thread-level parallelism is not very different from the mechanisms used for ILP. The processor must provide means for control flow speculation and high bandwidth instruction fetch, with the added requirement that both must be done for multiple programs.

Control flow speculation: To support TLP, control flow speculation is required for each thread, which can be achieved by building multiple next-block predictors, one for each thread, or simply sharing one predictor between multiple threads. In the TRIPS design, we share the next block predictor between the threads. Our performance analysis showed that good global exit history was crucial to the predictor accuracy. Sharing other tables like the local history and the predictor logic itself did not hinder multithreaded performance. So we replicated the global history shift registers and maintain one copy for each thread. The value in this shift register along with the program counter of that particular thread is used to make a prediction using the shared exit predictor tables. Since the global history registers amount to only 40 bits of storage (10 bits per thread), the resulting replicated storage is quite small.

High-bandwidth instruction-fetch: The management of the instruction caches and the network to stream instructions to the processor is again identical to what is required for supporting ILP. The only difference being that fetches of blocks are initiated from different threads every cycle, which is dependent on the rate at which threads complete. Tullsen et al. investigate several policies that can implemented for instruction fetch between multiple contending threads [163]. In the TRIPS prototype we implemented a simple round-robin scheme which gives equal priority to all executing threads and guarantees forward progress for every thread.

5.5.3 Data Storage Management

Short-term data: The management of short-term data is identical to what is done to extract ILP, since within each thread the processor extract ILP but to a lesser extent. The microarchitecture's naming convention of operands is such that these short-term data values passed between nodes in the dataflow graphs can never be sent to values from one thread to another thread.

Long-term data: To support multiple threads executing on the same processor core, sufficient replicated register storage must be provided to maintain the architecture state of each executing thread. One copy of the architecture registers is provided for each thread. Furthermore, the register renaming hardware must be aware that values should not be forwarded across threads, which is achieved by changes to only the control logic of the register renaming hardware. While no replication of temporary storage or datapath is required to create this reconfigurable register tile, one could argue that replicated register file storage is expensive and not in the spirit of polymorphism.

Persistent-data: The memory system operates much the same as when extracting ILP. Similar to modifications to the register renaming logic, the control logic in the data tiles is modified to ensure that load/store checking is performed only within a thread and not across threads.

5.6 Data-Level Parallelism

Data-level parallelism is most commonly found in streaming media and scientific applications and is characterized by the following main attributes: predictable loop-based control flow with large iteration counts, large data sets, regular access patterns, poor locality but tolerance to memory latency, and high computation intensity [155]. The dataflow graph abstraction already lends itself to efficiently supporting this kind of parallelism, since the concurrency is explicit in the ISA, compared to implicit parallelism expressed by RISC or CISC ISAs. We build polymorphous mechanisms to further optimize for the regular control and dataflow behavior exhibited by these applications.

In chapter 8 we present a detailed characterization of DLP programs and a derivation of mechanisms based on these attributes. In this section we discuss the bottlenecks of DLP programs in a conventional an ILP-like execution environment. Since, in principle, programs with DLP can be executed on the TRIPS processor relying on control flow speculation and having the hardware extract only ILP, this analysis uncovers the opportunities and potential for DLP specialization through polymorphism.

5.6.1 Execution Core Management

For programs with ILP and TLP, the dataflow graphs are typically small and control-flow speculation or explicit multithreading is necessary to generate a large window of potentially useful instructions. For programs with DLP, the compiler can construct large dataflow graphs by unrolling tight loops with large iteration counts.

As a result, the hardware overheads of speculation and software overheads of multithreading can be significantly reduced or completely removed. Instead, the most efficient way of managing the execution core to extract DLP is to unroll the graphs as much as possible and map large unrolled dataflow graphs to the reservation stations, without relying on speculation.

5.6.2 Control Flow Management

Control flow speculation is relatively less important for DLP programs, with power efficiency in instruction fetch and high bandwidth instruction fetch being more important. The SIMD execution paradigm is very efficient at amortizing instruction control management overheads across a large number of instructions and reducing design complexity, for exactly these type of programs. Polymorphous mechanisms can be used to tailor an architecture to achieve the efficiency of the SIMD model with only moderate changes to the instruction control logic. Executing the same dataflow graph in a loop with many iterations can be viewed as SIMD execution, where the dataflow graph can be viewed as one single SIMD instruction executed across multiple ALU sites. The overheads of repetitive instruction fetch and unnecessary speculation must be removed to reach the efficiencies that a true SIMD model can provide. We develop a mechanism called *instruction revitalization* that augments the instruction selection logic at each individual ET to reuse mapped instructions and augment the fetch logic to fetch instructions in a loop just once.

Also, with some types of DLP programs, a fine-grained multithreaded model that provides a MIMD execution model is preferred. The ILP and TLP execution model of sequencing a program counter that fetches and maps successive dataflow graphs (sometimes through control speculation) is not very efficient compared to this approach because they do not exploit control regularity. By adding instruction storage support and sequencing the ALUs independently the execution core can be tailed to look like a MIMD array and achieve its instruction fetch efficiencies.

5.6.3 Data Storage Management

Memory accesses in DLP programs are dominated by regular patterns, typically unit or fixed stride. However, significant numbers of other types of data accesses are also present, including irregular accesses to small lookup tables and accesses to a large number of run-time constants (coefficients of an FIR filter for example). This combination of structured and unstructured access patterns requires a data storage system that can provide high bandwidth regular data and low latency irregular operands.

Short-term data: The management of short-term data is identical to what is done to extract ILP. The large size of graphs typical when programs have DLP does not make any difference to the way most of these operands are managed. The strided regular memory accesses in these programs present an opportunity for optimizing some short-term data accesses. When performing regular memory accesses, individual load and store instructions that implement this strided access in the dataflow graphs, show regularity as well in the addresses these instructions generate. Such behavior is optimized in vector instruction sets by using some form of a load instruction that can read multiple words of data from memory and writing to a vector register file. Similarly, a multi-word load instruction can be used to fetch multiple words from memory and sending the operands to reservation stations in the ETs. Thus encoding strided access and amortizing the per-memory instruction overheads which include the execution overheads of multiple load instructions, the communication overheads of routing multiple address to the caches, and the memory access overheads of reading each word from the caches.

Long-term data: Accessing register values can become a bottleneck, if one register value has a high degree of fanout. For programs with ample DLP this is a commonly observed phenomenon. Furthermore, the programming model of sequentially executing dataflow graphs, with register values read for each dataflow graph introduces inefficiency when the register values do not change across each dynamic instance of the dataflow graph executed. For programs with DLP this type of read-only behavior can be determined by the compiler, whereas it can be more challenging for all programs. We propose a mechanism called *operand revitalization* whereby operands that do not change during multiple iterations of a dataflow graph are read once and reused multiple times, instead of being repeatedly read from the register file, incurring the overheads of register read and rename. This mechanism is not restricted to DLP, and can be utilized while extracting ILP or TLP if the compiler can statically determine this behavior.

Persistent data: To support DLP, a software managed cache memory built using the on-chip memory tiles is better than hardware managed conventional caching. Other designs like Smart Memories, Imagine, and the Cell processor have adopted this approach. To behave as a software managed memory, the reconfiguration of the memory tiles includes turning off tag checks to allow direct data array access and augmenting the cache line replacement state machine to include DMA-like capabilities. Enhanced transfer mechanisms include block transfer between the tile and remote storage (main memory or other tiles), strided access to remote storage (gather/scatter), and indirect gather/scatter in which the remote addresses to access are contained within a subset of the tile's storage. Instead of using the processor to orchestrate these transfers, a user-level DMA controller integrated on chip can perform these functions more efficiently.

5.7 Discussion

In this section, we described the principles of polymorphism and a core set of fundamental mechanisms to support instruction-level, thread-level, and data-level parallelism. Granularity of parallelism is fundamental to program behavior and we identify it as the first order difference between application types and characterize how it affects the microarchitecture.

The dataflow graph is used as a unifying abstraction to express concurrency for all three granularities of parallelism. For ILP, the processor resources are efficiently used to hold speculative instructions, with a next-block predictor (a specialized resource) used to perform control flow prediction. For TLP, which is coarse-grained concurrency across multiple threads, the processor resources are divided up between the threads and polymorphous control logic in the processor core ensures all threads get to use the processor datapath resources in a fair fashion. For DLP, which is characterized by concurrent operations on data, we identified the overheads of ILP style execution in this chapter. Chapter 8 includes a detailed analysis of DLP program behavior and the specification of polymorphous mechanisms for DLP.

To summarize, polymorphism serves as a natural way to address processor complexity and technology constraints and achieves design convergence while supporting different granularities of parallelism. The simplicity in implementation of the mechanisms and economy of these mechanisms suggests polymorphous architectures can be an attractive future computing substrate to build scalable architectures to support future application needs. In the following chapters we evaluate the performance that can be attained using these polymorphous mechanisms.

Chapter 6

Performance Evaluation: ILP

One of the primary goals of the TRIPS architecture and the ISA is to extract large amounts of concurrency. In this chapter we focus on instruction-level parallelism and demonstrate that the TRIPS processor has the *potential* to exploit greater concurrency than the best-of-breed ILP processors. Our evaluation is based on the prototype design using a cycle-accurate simulator which we have validated to be within 10% of the hardware.

We use a set of benchmark suites with different levels of complexity and different types of behavior to quantitatively evaluate the TRIPS design and demonstrate its effectiveness. We start with a set of hand-written microbenchmark kernels which we heavily hand optimized and tuned based on profiling the kernels and understanding the interactions between the code and the microarchitecture. This microbenchmark analysis demonstrates the potential of the architecture. We then employ a set of data parallel kernels and the EEMBC embedded benchmark suite to explore the performance of programs that are easy for the compiler to analyze. The control flow behavior of the DLP kernels and the EEMBC programs is quite regular and the memory footprint of many of the benchmarks is small. Finally, we evaluate the performance of the SPEC CPU2000 suite, whose programs are significantly more complex than the EEMBC benchmarks.

In Section 6.1 we describe the methodology of this ILP study and tools used in our performance evaluation. Section 6.2 describes the benchmarks. Section 6.3 discusses the performance results.

6.1 Methodology

To evaluate the performance of the TRIPS processor in advance of the manufactured chip, we developed a detailed cycle-level simulator, called *tsim-proc*, which models the hardware at a much more detailed level than higher-level simulators like SimpleScalar [30]. Our performance validation effort showed that performance results from *tsim-proc* were on average within 10% of those obtained from the RTL-level simulator, across a large number of crafted and randomly generated test programs. We use a critical path analysis tool (*tsim-critical* [115]) to attribute percentages of the critical path of the program to different microarchitectural activities using the technique first proposed by Fields et al. [52]. These results provide insight into the effectiveness and overheads of different components of the microarchitecture. To place the TRIPS processor in the context of a conventional microarchitecture, Table 6.1 lists its microarchitecture parameters.

Our baseline comparison point is a 467MHz Alpha 21264 processor, with all programs compiled using the native Gem compiler with the "-O4 -arch ev6" flags set. We chose the Alpha because it has an aggressive ILP core that still supports low FO4 clock periods, an ISA that lends itself to efficient execution, and a good compiler that generates extraordinarily high-quality code. We use Sim-Alpha, a

Processor parameter	**Configuration**
L1 Instruction Cache	Five 16KB banks, 2-way set associate, 1 port per bank
L1 Data Cache	Four 8KB banks, 2-way set associate, 1 port per bank
Registers	4 register banks, 32 registers per banks, 1 port per bank
Instruction Fetch	16 instructions per cycle
Instruction Issue	16 instructions per cycle
Instruction Commit	16 instructions per cycle
Load and Store ports	4 effective load and store ports
Control Flow Prediction	Predictor using exit histories to predict the next block, employing a tournament local/gshare predictor similar to the Alpha 21264 with 9K, 16K, and 12K bits in the local, global, and tournament exit predictors, respectively
L2 Cache	1 MB L2 cache, with 5 ports

Table 6.1: TRIPS processor parameters

simulator validated against the Alpha hardware to take the baseline measurements so that we could normalize the level-2 cache and memory system and allow better comparison of the processor and primary caches between TRIPS and Alpha [42].

6.2 Benchmarks

Since a key goal in this dissertation is to explore techniques to adapt one architecture to different types of workloads, we chose programs from different suites and application domains for this architecture evaluation study. The goal is to cover different granularities of parallelism, types of instruction mixes, and basic program behavior. We use four separate suites of benchmarks: 1) a set of hand-tuned heavily optimized microbenchmarks, 2) a set of kernels we developed with ample data-

List of Benchmarks	
Microbenchmarks: sha, dct8x8, matrix, vadd	
Data Parallel Benchmark Kernels	
Scientific Computing	LU, FFT
DSP	convert, dct, fir
Graphics Processing	3 vertex shaders and 2 fragment shaders
Network Processing	AES, MD5, and Blowfish
EEMBC Benchmarks: All 30 benchmarks	
SPEC CPU2000	
Integer	Floating Point
164.gzip	168.wupwise
175.vpr	177.mesa
181.mcf	179.art
197.parser	200.sixtrack
256.bzip2	301.apsi
300.twolf	

Table 6.2: List of benchmarks

level parallelism (DLP), 3) the EEMBC suite [47], and 4) the SPEC CPU2000 suite [153]. Table 6.2 lists the benchmarks which are described below.

Microbenchmarks: To demonstrate the effectiveness of the architecture without being hampered by compiler technology, we use four separate microbenchmarks that are very specific in their behavior. *sha* is a hashing algorithm and is a very sequential program with limited amounts of concurrency. `dct8x8` is an 8x8 optimized discrete cosine transform computation that uses only integer math. *matrix* is a straight-forward matrix multiplication program. *vadd* does vector addition of two 2048-element vectors. All of these kernels are quite small and are possible to hand-optimize based on feedback obtained from simulation and critical path analysis.

DLP kernels: We developed the data parallel benchmarks to understand DLP program behavior to drive our exploration of polymorphous mechanisms for data-level parallelism. For the sake of continuity we present the rationale, the development process, and detailed description of the benchmark suite when we analyze DLP behavior in chapter 8 and we include just a brief summary here. The DLP kernels cover a large, if not entire, space of data parallel applications and are grouped into four broad categories with a total of 13 kernels.

EEMBC and SPEC CPU2000: We used all 30 of the EEMBC benchmarks which are split into five categories called: automotive, consumer, networking, office, and telecom. They are all heavily loop based with small working set sizes and instruction footprints. We adjusted the iteration counts of the EEMBC benchmarks to reduce their execution time and hence simulation time. We used a subset of SPEC CPU2000 benchmarks for which the reduced input set sizes made simulation tractable. We used the reduced input set sizes distributed as part of the MinneSPEC workloads [91].

All these benchmarks were compiled using the TRIPS compiler toolchain which takes C or FORTRAN77 code and produces complete TRIPS binaries that will run on the hardware. Although the TRIPS compiler is able to compile these major benchmark suites correctly [146], many TRIPS-specific optimizations are currently being developed and incorporated into the compiler. Prior to completion of those optimizations, the TRIPS compiler will be inadequate to evaluate the architecture because many of the TRIPS blocks are too small.

Benchmark	Speedup TCC/Alpha	Speedup Hand/Alpha	IPC Alpha	IPC TCC	IPC Hand
dct8x8	2.25	2.73	1.69	5.13	4.78
matrix	1.07	3.36	1.68	2.85	4.12
sha	0.40	0.91	2.28	1.16	2.10
vadd	1.46	1.93	3.03	4.62	6.51

Table 6.3: TRIPS performance results on microbenchmarks.

6.3 Results

6.3.1 Microbenchmarks

Table 6.3 shows the performance of the TRIPS processor compared to the Alpha for the four microbenchmarks. This study with the microbenchmarks is intended to demonstrate the capabilities of the microarchitecture and reveal bottlenecks in the architecture.

The second column shows the speedup of TRIPS compiled code (TCC) over the Alpha. We computed speedup by comparing the number of cycles needed to run each program on the two simulators. The third column shows the speedup of the hand-generated TRIPS code over that of Alpha. Columns 4–6 show the instruction throughput (instructions per cycle or IPC) of the three configurations. The ratio of these IPCs do not correlate directly to performance, since the instruction sets differ, but they approximate the level of concurrency each machine is exploiting. The disparity between the compiled and hand-optimized TRIPS code indicates the current inefficiencies in the compiler.

The results show that for the hand optimized programs, the TRIPS distributed microarchitecture is able to sustain reasonable ILP, ranging from 2.1 to 6.5.

The speedups over the Alpha core range from 0.9 to 3.36. **sha** sees a slowdown on TRIPS because it an almost entirely serial benchmark. What little concurrency there is, is mined out by the Alpha core. The wider TRIPS core provides no additional benefit, and instead the TRIPS processor performs slightly worse because of the block overheads, such as inter-block register forwarding. **vadd** has speedup close to two because the TRIPS core has exactly double the L1 memory bandwidth that the Alpha does (four ports as opposed to two), resulting in an upper-bound speedup of two. These results demonstrate the potential of the TRIPS core and show that it is possible to build a ultra-wide issue distributed processor to efficiently mine concurrency in sequential programs.

The compiler-generated version of these microbenchmarks do not perform as well as the hand-optimized version. For *matrix* and *vadd* the compiler generated code is not unrolled optimally and the contention for routing loads and stores to the memory system becomes a significant bottleneck. For *sha* the compiler does not effectively predicate the code sufficiently to create large hyperblocks. While the compiler-produced results are far from the best we expect to obtain, they do give some insight into the capabilities of TRIPS. The hand optimized kernels demonstrate what the architecture is capable of, if the compiler can be made sophisticated enough to match such hand optimizations.

6.3.2 Data Parallel Kernels

Table 6.4 shows the performance obtained on the data parallel benchmark suite. These applications have ample DLP and are typically coded in specialized

Benchmark	TRIPS TCC			Alpha		Speedup
	IPC	Cycles (1000s)	Block size	IPC	Cycles (1000s)	
DSP/convert	6.05	54	61	0.6	5	0.11
DSP/dct	4.27	58	61	2.1	87	1.49
DSP/highpassfilter	6.94	677	81	1.8	1613	2.38
graphics/fragmentreflection	1.83	616	31	0.9	294	0.48
graphics/fragmentsimplelight	2.44	759	28	0.6	366	0.48
graphics/vertexreflection	2.74	505	33	1.1	358	0.71
graphics/vertexsimplelight	2.35	881	30	0.8	489	0.56
graphics/vertexskinning	4.10	446	55	1.3	918	2.06
network/blowfish	1.20	1168	18	1.7	465	0.40
network/md5	0.76	2225	7	1.4	460	0.21
scientific/LU	0.69	20770	80	1.0	11181	0.54
scientific/fft	1.36	17	22	1.4	21	1.19

Table 6.4: Processor performance on DLP kernels

ISAs. For example, the graphics kernels will be coded in the assembly language of the vertex shader or fragment shader processor in a graphics chip. However, for the purpose of this evaluation, they are written in C, assuming a sequential programming model and compiled using the TRIPS toolchain to produce block atomic TRIPS binaries. No hand optimization or architecture specific tuning of the source code was performed for these experiments. This benchmark suite has more sophisticated behavior than the set of microbenchmarks discussed previously and is representative of real DLP workloads.

The programs in this suite are highly concurrent and as shown in the second column in Table 6.4 the processor is able to extract significant amount of ILP - the IPCs range from 0.6 to 6.4. One of the reasons for the high performance is that the compiler mostly generates programs with large blocks, as shown by the average dynamic block sizes in the third column, which varies from from 7 to 81. We now briefly analyze these results grouping the benchmarks according to common behavior.

Low ILP: The three network processing benchmarks are outliers as they show low IPCs. The network processing benchmarks perform a significant amount of computation for every network packet, each of which typically consists of 1500 bytes of data. The computations include algorithms for encryption and hashing, which are typically serial in nature (similar to the *sha* microbenchmark). However, packet processing applications offer other means of concurrency such as processing packets in parallel, or processing independent streams of packets in parallel. In the sequentially coded version of the program the compiler or the hardware is unable to reach the parallelism that is available across such distant regions in the program and the only concurrency that can be mined is ILP in the dynamic instruction window. In chapter 8 we discuss how to tailor the hardware to look like a decoupled execution array to mine more concurrency in such scenarios.

Memory intensity: The two scientific processing kernels, *fft* and *LU*, are similar in that they make heavy use of the memory system. Although the block sizes that the compiler can generate are quite large (79 and 22), the final IPC during program execution is quite low – around 1. Both *f*ft and *L*U have a large number of memory accesses. Unfortunately, because the scheduler is unaware of the memory addresses of loads and stores in each block, it is unable to place these instruction in such a way that their contention for the TRIPS operand network links is low. The *vadd* microbenchmark shows similar behavior–the compiler generated code was 66% worse than hand optimized code.

High ILP: Most of the programs have high ILP with IPCs as high as 6.94. Using dataflow graphs and building a large dynamic instruction sequence through control flow speculation is effective at exposing data-level parallelism to the hardware. Alternate approaches of vectorization or SIMD computation that are meant for DLP computation are likely to perform better. In chapter 8 we describe our experiments that compare the performance of specialized data parallel architectures to polymorphous DLP mechanisms.

6.3.3 EEMBC and SPEC CPU2000 Benchmarks

Tables 6.5 and 6.6 show the performance obtained on the EEMBC and SPEC CPU2000 benchmarks. Most of the EEMBC benchmarks are very regular, with small data set sizes, whereas the SPEC benchmarks are more representative of general purpose workloads. The IPC across these benchmarks is much lower than what we observed in the previous two suites - the values range from 0.53 to 2.31. Most of the benchmarks perform worse on TRIPS than on Alpha–only 9 of the 30 EEMBC benchmarks perform better, and only 2 of the 11 SPEC CPU2000 benchmarks perform better on TRIPS. One of the main reasons for the lower performance is that the average block sizes that the compiler is able to construct is much smaller for these benchmarks. In addition, the control misprediction rate is higher in the SPEC benchmarks as these have more irregular control flow than the simple DLP

Benchmark	**TRIPS TCC**			**Alpha**		**Speedup**
	IPC	**Cycles (1000s)**	**Block size**	**IPC**	**Cycles (1000s)**	
automotive/a2time01	0.50	226	8	1.0	404	1.79
automotive/aifftr01	1.32	7506	39	1.4	9793	1.30
automotive/aifirf01	0.63	262	11	1.4	99	0.38
automotive/aiifft01	1.29	7094	43	1.6	8237	1.16
automotive/basefp01	0.63	288	11	0.8	238	0.83
automotive/bitmnp01	1.34	932	32	0.9	1055	1.13
automotive/cacheb01	0.66	746	22	0.9	391	0.52
automotive/canrdr01	0.91	1485	26	1.2	805	0.54
automotive/idctrn01	1.37	521	23	1.5	610	1.17
automotive/iirflt01	0.71	603	21	1.2	507	0.84
automotive/matrix01	1.00	7782	40	1.4	4578	0.59
automotive/pntrch01	0.82	1621	29	0.8	1183	0.73
automotive/puwmod01	0.91	2262	30	1.3	1199	0.53
automotive/rspeed01	0.93	785	22	1.1	535	0.68
automotive/tblook01	0.60	332	12	1.1	108	0.33
automotive/ttsprk01	0.86	1073	26	1.3	669	0.62
consumer/cjpeg	1.58	49549	31	1.2	61498	1.24
consumer/djpeg	1.30	78197	34	1.3	68276	0.87
networking/ospf	0.98	3515	26	1.2	2167	0.62
networking/pktflow	1.16	10088	24	1.4	6305	0.62
networking/routelookup	0.93	7395	30	1.2	4097	0.55
office/bezier02	1.22	3216	25	1.1	7332	2.28
office/dither01	1.83	8647	48	1.8	7835	0.91
office/rotate01	1.42	5890	41	1.4	3302	0.56
office/text01	1.08	9401	23	1.3	5413	0.58
telecom/autocor00	0.53	273	8	1.1	60	0.22
telecom/conven00	1.82	1389	23	2.1	993	0.72
telecom/fbital00	1.58	2173	38	1.9	3267	1.50
telecom/fft00	2.85	2327	33	1.6	6548	2.81
telecom/viterb00	1.20	2727	33	1.8	2711	0.99

Table 6.5: Processor performance on EEMBC benchmarks

Benchmark	TRIPS TCC			Alpha		Speedup
	IPC	Cycles (millions)	Block size	IPC (millions)	Cycles	
fp/168.wupwise	1.90	2940	28	1.4	3490	1.19
fp/177.mesa	2.00	5038	50	0.8	8273	1.64
fp/179.art	2.15	2179	42	0.9	1880	0.86
fp/200.sixtrack	0.92	2549	12	1.2	1178	0.46
fp/301.apsi	2.31	89	40	1.5	47	0.53
int/164.gzip	1.57	1823	23	1.4	994	0.55
int/175.vpr	1.14	30	24	1.2	14	0.46
int/181.mcf	1.90	244	28	1.1	126	0.52
int/197.parser	1.00	568	12	1.3	191	0.34
int/256.bzip2	1.49	2271	21	1.4	1288	0.57
int/300.twolf	0.84	212	22	1.0	85	0.40

Table 6.6: Processor performance on SPEC CPU2000 benchmarks

benchmarks and microbenchmarks.

In general these programs are much more influenced by the level of sophistication in the compiler, as they are built from large code-bases and rely on function inlining, sophisticated loop transformations and predication heuristics to build large hyperblocks. Second, their dynamic behavior in terms of memory accesses, contention caused in the operand network, load-store dependence conflicts, and control speculation all vary significantly and can cause performance losses. In spite of these drawbacks, our results show moderate amounts of concurrency being exploited by the core. Since the code quality from our compiler is not very good, most of these benchmarks perform worse on TRIPS than on Alpha.

6.4 Summary

We conclude from this analysis that the TRIPS microarchitecture can sustain good instruction-level concurrency, despite all of the distributed overheads, given kernels with sufficient concurrency and aggressive handcoding. Whether the core will be able to exploit ILP on complete benchmarks, or whether the compiler will be able to generate sufficiently optimized code, remain open questions that are subjects of ongoing work in the TRIPS project. Even so, compiled TRIPS code performs competitively compared to the Alpha on many microbenchmarks. On complex programs like the SPEC CPU2000 benchmarks, the TRIPS processor performs worse than the Alpha, since the code quality generated by our compiler on these programs is poor. The maturation time of a compiler for a new processor is not short, but we anticipate significant improvements as our hyperblock generation and optimization algorithms come online.

The polymorphism mechanisms that support ILP are the high bandwidth instruction fetch, the reservation stations that are managed as a large instruction window, the next-block predictor and the LSQ logic. Although, the next-block predictor and the LSQ logic are heavily tuned to extracting ILP, we show in the next chapter how they provide performance improvement while extracting TLP also, by providing support for small levels of ILP within each thread.

There are several novel features in this ISA, execution model, and microarchitecture. Evaluating these aspects in detail is beyond the scope of this work, and Nagarajan provides a detailed analysis covering many of these topics in his dissertation [114]. Novel features in the ISA that are studied include fanout optimizations and predication optimizations. The different micronet protocols and their overheads are the two main features of the microarchitecture that can affect performance and a detail critical path analysis of different microarchitecture events shows the bottlenecks in the design.

In this chapter, we have focused on demonstrating the potential for the architecture and making the case for this class of ISAs and partitioned microarchitectures from a performance standpoint. These results show that the architecture can perform well on a broad class of programs and can excel on hand optimized programs. It serves as our starting point for evaluating polymorphism to see how TRIPS can be configured using polymorphism to match specialized processors across a broad class of applications.

Chapter 7

Performance Evaluation: TLP

In this chapter, we evaluate the performance of polymorphous mechanisms for TLP implemented in the TRIPS prototype. We briefly outline the methodology used for obtaining these results and then discuss the performance results. The polymorphous mechanisms to support thread-level parallelism include the following.

Execution core: The reservation stations in the execution core are partitioned between multiple threads. The TRIPS prototype chip implements a static partitioning approach in which each thread can utilize up to 256 of the available 1024 reservation stations. Since each block requires 128 reservation stations, one speculative and one non-speculative block can execute simultaneously for each active thread. Up to 4 independent programs can execute concurrently on the processor.

Control flow: Polymorphous mechanisms are implemented in the block fetch logic and next block predictor. The block fetch logic is augmented to cycle between the different program threads as they commit their blocks and fetch slots become empty. Next block prediction is provided for each thread with a separate 12-bit global history register for each thread. The other storage structures in the next-block predictor which include the branch target buffer, call target buffer, and the return address stack are shared between all threads.

Data storage: The register tiles have support for performing register renaming only between blocks that belong to one thread. The data tiles include support for checking for load/store dependence between memory instructions in a single thread.

Other: Finally, the processor has two special registers called the Thread Control Register (TCR) and Processor Control Register (PCR) that can be used to configure the processor. The PCR register can be set to configure the processor into a multithreaded mode and the TCR register can be used to set the number of threads that must execute.

In this dissertation, we refer to this multithreaded mode as the TLP-mode of the processor, while other publications have used the term *T-morph* to refer to this mode. While evaluating the TLP mechanisms, we compare execution time to a configuration where each program is run separately on the processor with all processor resources devoted to extracting only ILP from that single program. In the remainder of this chapter we refer to such an execution configuration as the ILP-mode of the processor. For the purpose of consistency in writing, this dissertation uses this terminology of *ILP-mode*. Previous publications have referred to such a configuration as the *D-morph* mode of the processor.

7.1 Methodology

The cycle-accurate simulator *tsim-proc* described in the previous chapter also models the polymorphous mechanisms for TLP. We used this simulator for the results presented in this chapter. The compilation strategy used and the binaries are identical to the ILP study described in the previous chapter. All the benchmarks used were compiled using the TRIPS compiler toolchain which takes C or FORTRAN77 code and produces complete TRIPS binaries. We adjusted the iteration counts of the EEMBC benchmarks to reduce their execution time and hence simulation time. We used a subset of SPEC CPU2000 benchmarks for which the reduced input set sizes made simulation tractable.

7.1.1 Configurations

We study three processor configurations which are listed in Table 7.1. In all configurations $1/4th$ of next-block predictors storage tables are provided to each program with separate 10-bits of global history devoted to each program. The 1-Thread configuration and the 2-Thread configuration leave $3/4th$ and half of the processor storage resources un-utilized, respectively. This is an artifact of the static resource partitioning decision that was made for the prototype implementation and does not imply the polymorphous mechanisms cannot fully utilize the processor resources when fewer than 4 threads are available.

Configuration	Description	Resources
1-Thread	One single thread running in the processor, with the processor configured to run in TLP-mode. For this thread there is never more than one speculative block executing. When executing in the baseline ILP-mode of the processor, in comparison, there can be up to eight speculative blocks executing.	1. 256 reservations stations allocated to one program, 768 of 1024 reservation stations unused. 2. 128 physical registers allocated to one program. 384 physical registers unused.
2-Thread	Two threads executing with each thread having not more than one speculative block executing.	1. 256 reservations stations allocated to each program, 512 of 1024 reservation stations unused. 2. 128 physical registers allocated to each program. 256 physical registers unused.
4-Thread	Four threads executing with each thread having not more than one speculative block executing.	1. 256 reservations stations allocated to each program, none of 1024 reservation stations unused. 2. 128 physical registers allocated to each program. No physical registers unused.

Table 7.1: Different processor modes simulated

7.1.2 Workload

We execute different mixes of programs in both the 2-Thread configuration and 4-Thread configuration. A key methodological question to address is what type of program mixes to chose for such a study. Previous researchers have classified programs using different criteria such as memory behavior characterized by L2 cache miss rates, control speculation behavior characterized by branch prediction accuracy, instruction footprint characterized by L1 instruction cache miss rates and combined applications with similar and dis-similar characteristics to study the sensitivity of the architecture to the workload.

In previously published work, we adopted this approach to evaluate a subset of the SPEC CPU2000 benchmarks by creating such workload mixes [141]. We classified programs into two categories namely, *low memory intensive* and *high memory intensive* based on the L2 cache miss rates and ran combinations of all 3 mixes: high/low, low/low, and high/high. Other features of programs that could affect execution efficiency in multithreaded mode include the available concurrency in the programs, control speculation accuracy, and operand network contention.

In this dissertation, we undertake a more thorough analysis of multithreaded execution. We have a large application space which includes 30 EEMBC programs, 11 SPEC CPU2000 programs, and 13 DLP kernels. It is hard to determine a-priori what application characteristics are important and isolate the phase behavior of these applications. For this study, we decided on the approach of using a large number of random program mixes and generated enough mixes to create different types of overlapping program behavior. By covering a significantly larger portion of the program behavior, this approach provides a more comprehensive evaluation of multithreading efficiency. This evaluation strategy is similar to the methodology used by Tullsen et al. and other publications on SMT [163].

We exclude the four microbenchmarks from this study, as they are primarily meant for demonstrating the potential of the processor, and do not form a meaningful benchmark suite for studying multithreading efficiency. Furthermore, some of the optimizations implemented in those benchmarks assume a single threaded execution mode with all 1024 reservation stations available to the program. All programs are run to completion and when a program finishes while others are still executing, it is restarted. When every program has completed execution once, we stop the simulation and collect simulation data. Since the EEMBC suite, SPEC CPU2000 suite, and the DLP kernels have very different behavior and run-times, we chose program mixes such that all the programs run as a multi-programmed workload were from the same suite.

7.1.3 Performance Metrics

The three performance metrics that we use for evaluation are:

1. **Processor Utilization:** The functional resources in the processor that are kept busy. We measure the number of instruction retired per cycle (IPC) to measure processor utilization. We compare the processor utilization between the TLP-mode and ILP-mode of the processor. In the ILP-mode we assume the programs in the workload mix are executed serially, and the IPC reported for the ILP-mode for that application mix is the total number of instructions

executed across all the applications in the mix divided by the total number of cycles taken.

2. **Processor Speedup:** The speedup compared to executing the mix of application in a serialized mode, executing one after another exploiting ILP only. Mathematically, where E is the execution time in cycles::

$$\text{Speedup} = \{\frac{\sum \text{For all programs } E_{ILP-mode}}{E_{TLP-mode}} - 1\} * 100$$

3. **Processor Efficiency:** Efficiency of the TLP-mode in overcoming resource and contention conflicts. We compare the execution of multiple threads on one single processor in TLP-mode, to executing each thread independently on its own dedicated TRIPS processor. We measure efficiency by comparing performance against two configurations, called *ideal* and *max*, both of which execute multiple programs concurrently on dedicated processors cores. The first configuration, *ideal* is the default ILP-mode of the processor in which up to eight speculative blocks can execute simultaneously utilizing all of the 1024 reservation stations in the processor. The second configuration, *max*, utilizes only a quarter of the reservation stations in the processors with at most one speculative block executing along with the non-speculative block. This configuration isolates the resource conflicts from the contention conflicts by creating an environment in which a program executes with the same set of resources it will have in the TLP-mode, but no contention from other threads. Mathematically, where E is the execution time in cycles:

$$\text{Efficiency}_{max} = \{\frac{E_{TLP-mode}}{\text{Max}(E_{\text{All programs in 1-Thread TLP-mode}})} - 1\} * 100$$

$$\text{Efficiency}_{ideal} = \{\frac{E_{TLP-mode}}{\text{Max}(E_{\text{All programs in ILP-mode}})} - 1\} * 100$$

Note that compared to the TLP-mode, both the *ideal* and *max* configuration use 2 full processors for executing 2 threads and 4 full processors when executing 4 threads. The *ideal* configuration is the limit performance possible and captures the overall efficiency of TLP execution and the TRIPS implementation of TLP support. The *max* configuration is maximum performance that can realistically be achieved given the physical resource constraints of the TRIPS TLP mode and captures the overheads of contention for shared resources.

7.2 Results

We discuss the performance results for each of the three suites, namely SPEC CPU2000, EEMBC, and DLP kernels, individually. Our workload consists of random mixes of programs, all picked from the same suite.

Figures 7.1 through 7.3 show results for the SPEC CPU2000 suite, Figures 7.4 through 7.6 show results for the EEMBC suite, and Figures 7.7 through 7.9 show results for the data-parallel benchmarks. Tables 7.2 through 7.7 show the program mixes that were executed.

7.2.1 SPEC CPU2000 Benchmarks

Utilization: Figure 7.1 shows the IPC for the 2-Thread and 4-Thread configurations with the workload mixes sorted by the difference between IPC in the TLP-mode and IPC in ILP-mode. For each program mix, the IPC when executing in TLP-mode is shown along with the overall IPC when the programs are executed serially in ILP-mode.

For the 2-Thread configuration, on average the IPC is 1.45 in the TLP mode which is approximately the same as the IPC in the ILP-mode. The range of IPCs are also similar, between 0.27 and 3.44. However, we can clearly see 4 distinct types of behavior. Recall that the main difference to a program's execution environment in the TLP 2-Thread configuration compared to the ILP-mode are: 1) reduced speculation depth, from 8-deep to 2-deep, 2) reduced instruction window, 256 entries per thread instead of 1024, and 3) contention for the shared resources like data tiles, operand network, and register files. The 4 types are:

1. **ILP-mode $>>$ TLP-mode (average 48% better) :** In 13 of 40 mixes, the ILP-mode of execution provides better processor utilization than the TLP-mode, more than 25% better. This poor performance of the TLP-mode is a result of the simple partitioning strategy which leaves half the processor's reservation stations unused when only two threads are executing . Each thread gets to execute one speculative block and one non-speculative block only. This drop in utilization is most dramatic for programs with good control predictability and high levels of concurrency. Specifically, four programs in this

suite, *fp/171.swim, fp/173.applu, fp/183.equake,* and *fp/172.mgrid* show an almost 2X drop in performance when the processor's effective window size is reduced from 256 to 1024 as shown in Appendix B. The 13 mixes corresponding to this case are dominated by these 4 benchmarks.

As a quick aside, we discuss Appendix B here. We compare the performance of a program executing in the ILP-mode mode to a 1-Thread TLP-mode. Recall that the 1-Thread TLP-mode is similar to the ILP-mode, but with only 256 reservation stations available to a program. Appendix B shows this performance comparison for the DLP, EEMBC, and SPEC CPU2000 benchmark suites.

2. **ILP-mode $>$ TLP-mode (average 17% better) :** Eight mixes, from 14 through 21 perform slightly better in the ILP-mode than the TLP-mode–up to 25% better. These are mixes where the programs have small amounts of ILP and not very good control speculation, so the reduction in control speculation depth does not significantly reduce performance. For these programs, blocks that are beyond a speculation depth of two do not provide significant amounts of useful work in the ILP-mode.

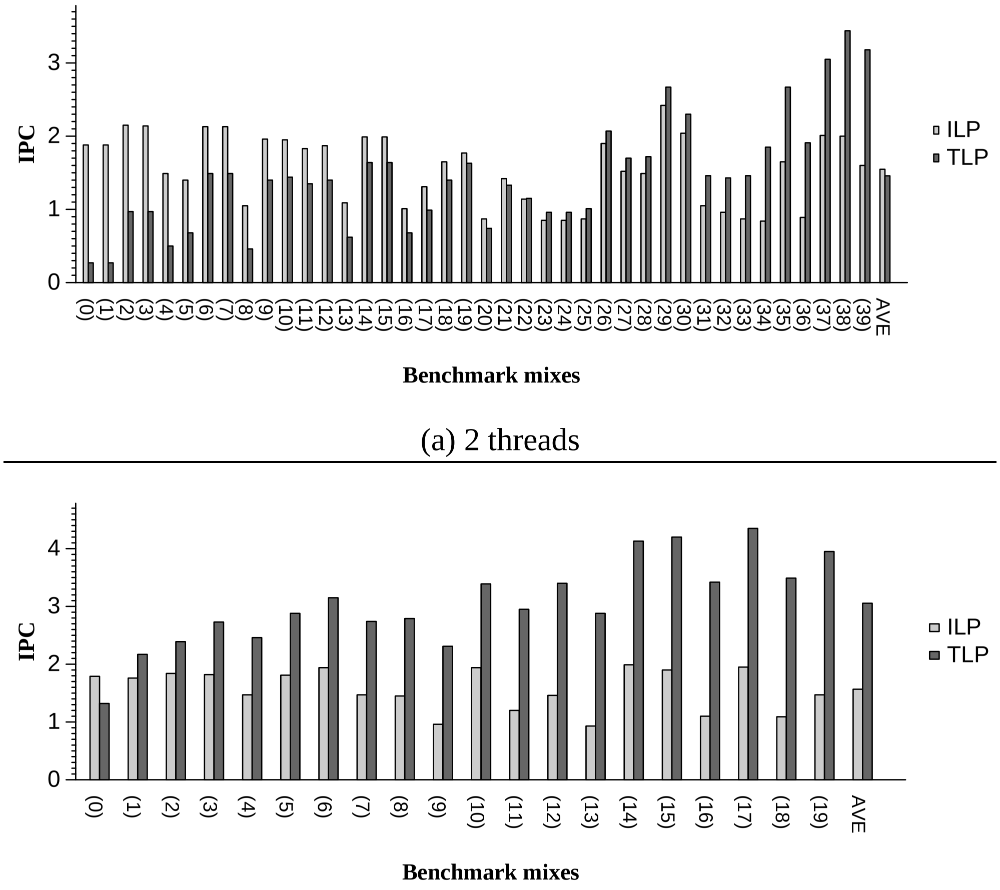
3. **TLP-mode $>$ ILP-mode (average 11% better) :** Mixes 21 through 30 perform slightly better in TLP-mode, up to 12% better. These are mixes where one application's performance is severely limited by the reduced instruction window, whereas another is not limited.

4. **TLP-mode $>>$ ILP-mode (average 75% better) :** Finally mixes 31 through

39 perform much better in TLP-mode than on ILP-mode, on average 68% better and as much as 2X better when *int/164.gzip* and *fp/301.apsi* execute together. These are mixes where the IPC of both applications is quite low to start with, and they have poor control speculation accuracy. As a result, reducing the size of the instruction window, and hence the control speculation depth, does not reduce performance significantly. Instead, the presence of two threads, and hence two sources of useful non-speculative work every cycle, improves the overall processor utilization.

The results show less diverse behavior in the 4-Thread configuration with the TLP-mode being worse for only one program mix. On average, the IPC is 3 and ranges from 1.32 to 4.35 which is significantly better than the ILP-mode IPC. The workload mix in which the TLP-mode does worse comprises of *fp/179.art, int/256.bzip2, fp/173.applu*, and *fp/188.ammp*. All four of these programs are very memory intensive and benefit significantly from control speculation. Firstly their performance difference between ILP-mode execution and the TLP-mode execution of only 256 reservation stations is high–ranges between 54% and 95%. Secondly, since they are memory intensive, the data tiles become a significant bottleneck while trying to execute these four programs concurrently.

For all other program mixes, the processor is able to overcome the contention effects of sharing resources between multiple threads quite effectively. Secondly, with four available threads the processor has a large amount of useful work, at least 4 useful blocks every cycle. In the TLP-mode, the benefits of having more useful non-speculative work overcome the inter-thread contention effects. To summarize, the polymorphous mechanisms are able to effectively utilize the processor when executing four threads. When executing two threads, the simple static partitioning approach results in wasted resources and as a result the TLP-mode has better utilization than the ILP-mode in only half of the program mixes. These results suggest a more sophisticated partitioning approach can help improve utilization still further when only a small number of threads are available.



(a) 2 threads

(b) 4 threads

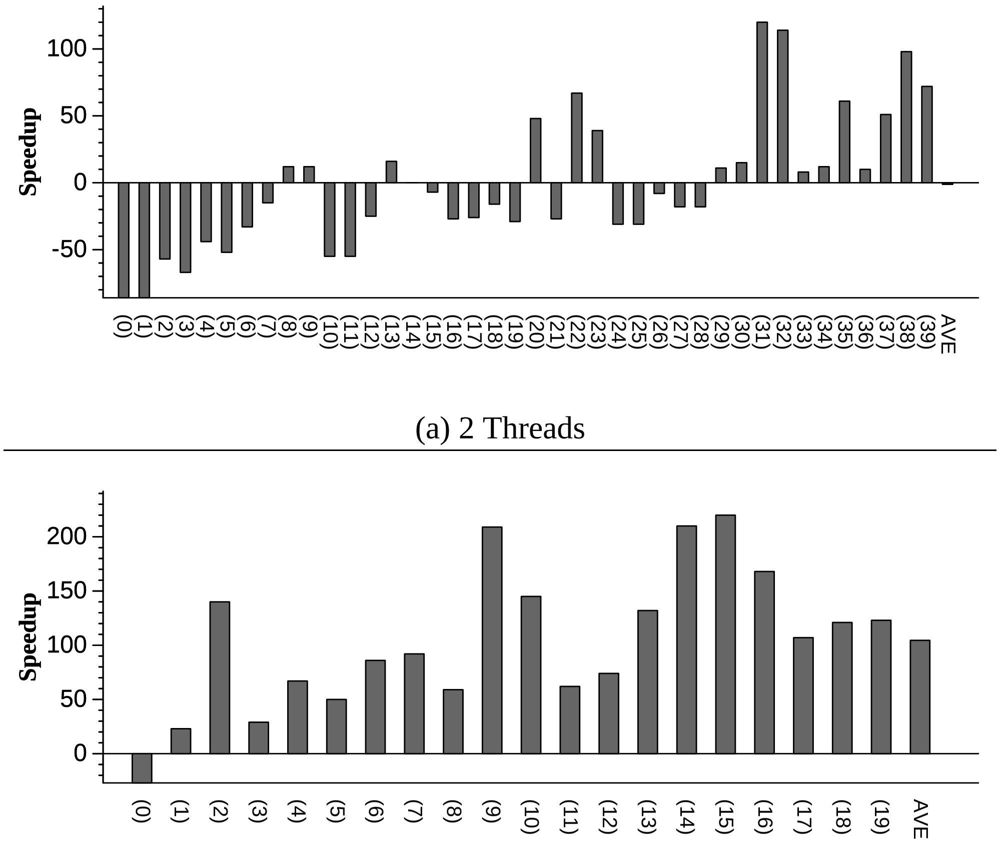
Figure 7.1: TLP-mode performance (utilization) - SPEC CPU2000 suite.

Speedup: Figure 7.2 shows speedup achieved by executing in TLP-mode, compared to serialized execution of the multi-programmed workloads in ILP-mode. The workload mixes are sorted in the same order as for Figure 7.1. In the 2-Thread configuration, of the 40 mixes, 18 show a slowdown (average 36% slowdown), and 22 show a speedup, up to 220%, and on average 43%. This speedup or slowdown exhibited by a program mix is primarily a function of the available parallelism in the programs. When there is a lot of parallelism in the threads, the 2-Thread configuration of the TLP-mode does not fully utilize the processor because only two simultaneous blocks from a single thread can be executing at a time (the effective instruction window size is 256), while in the ILP-mode the effective instruction window size is 1024. Hence, a slowdown in the TLP-mode is most likely to occur for programs with ample concurrency. For each of the program mixes, we examined IPC in the ILP-mode and saw that the average IPC of the programs in the mixes that exhibit a slowdown is 3.24, while that of the mixes that exhibit a speedup is 2.4. A more sophisticated partitioning of reservation stations between threads, allowing 512 entries per thread, is likely to improve this speedup.

While executing 4 threads, where the entire instruction window is utilized, with 256 entries assigned to each thread, only one program mix does worse in the TLP-mode compared to serial execution in ILP mode. On average the speed is close to 100% compared to the ILP-mode and ranges from 73% to 220%. The primary reason behind the speedup achieved by the TLP-mode, is that the effects of branch mis-speculation are lower than in the ILP-mode as a result of the reduced speculation depth per thread. In fact, examining the simulation statistics we saw that the average number of processor flushes in TLP-mode is less than half that compared to ILP execution. Not only is the processor executing programs faster in most cases, it is also spending fewer cycles in wasted speculative work.

Efficiency: We measure efficiency by comparing performance against two configurations, called *ideal* and *max*, both of which execute multiple programs concurrently. Figure 7.3 shows the efficiency of the TLP-mode for the 2-Thread and 4-Thread configuration. Recall that, while the *ideal* efficiency captures the overheads of multithreading implementation in the TRIPS chip, the *max* efficiency captures the overheads of contention alone.

In the 2-Thread configuration, on average an efficiency of 84% is achieved compared to the *max* configuration, implying the overheads of contention result in a 16% performance loss, compared to an oracle machine that completely hides this contention. The average *ideal* efficiency is 49%, implying the TRIPS implementation for TLP, has a 51% performance loss compared to an oracle machine that has no resource limitations for multithreading and can completely hide inter-thread contention. Of the 40 mixes, 4 mixes, namely, *10, 32, 34,* and *36* surprisingly show *ideal* efficiencies that exceed the *max* efficiency, and in the case of mix *34* the efficiency exceeds 100%. All of these mixes execute *int/254.gap* combined with one other program. Control speculation behavior for *int/254.gap* explains this non-intuitive behavior of more hardware resulting in poorer performance. Table B.1 in Appendix B shows that a *reduction* in speculation depth which is accompanied by a *reduction* in resources from 1024 to 256, *improves* performance by 65% for



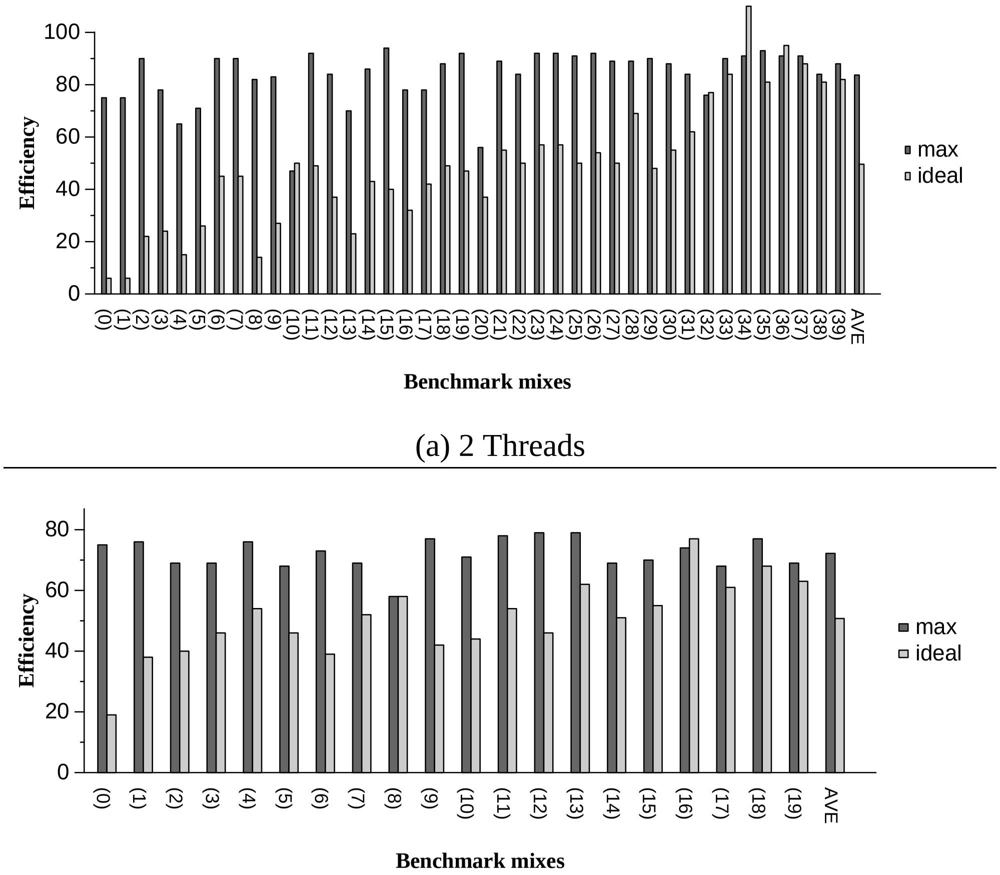
(a) 2 Threads

(b) 4 Threads

Figure 7.2: TLP-mode speedup compared to serialized execution - SPEC CPU2000 suite.

this program. As a result of this behavior, the *ideal* efficiency exceeds the *max* efficiency.

The efficiencies in the 4-Thread configuration are similar, 72% *max* efficiency and 50% *ideal* efficiency. There is little change in the efficiency because the increase in resources between the 4-Thread configuration and the configuration we are comparing to *ideal* and *max* is the same. The number of reservation stations increased from 512 to 1024 in the former, while the total number of processors increased from two to four in the latter. Program mix *16* again exhibits the anomalous behavior of higher ideal efficiency compared to max efficiency because it contains two copies of *int/254.gap*.



(a) 2 Threads

(b) 4 Threads

Figure 7.3: TLP-mode execution efficiency - SPEC CPU2000 suite.

(0)	fp/171.swim, fp/173.applu
(1)	fp/173.applu, fp/171.swim
(2)	fp/179.art, fp/173.applu
(3)	fp/171.swim, fp/179.art
(4)	fp/171.swim, int/256.bzip2
(5)	fp/183.equake, int/175.vpr
(6)	fp/179.art, int/175.vpr
(7)	fp/179.art, int/175.vpr
(8)	fp/173.applu, int/197.parser
(9)	fp/172.mgrid, int/181.mcf
(10)	fp/177.mesa, int/254.gap
(11)	fp/168.wupwise, int/300.twolf
(12)	fp/171.swim, int/181.mcf
(13)	int/300.twolf, fp/171.swim
(14)	fp/171.swim, fp/177.mesa
(15)	fp/173.applu, fp/177.mesa
(16)	fp/188.ammp, int/175.vpr
(17)	int/186.crafty, fp/183.equake
(18)	fp/188.ammp, int/181.mcf
(19)	fp/168.wupwise, int/164.gzip
(20)	int/300.twolf, int/175.vpr
(21)	int/186.crafty, int/256.bzip2
(22)	int/175.vpr, int/175.vpr
(23)	int/300.twolf, int/255.vortex
(24)	int/255.vortex, int/300.twolf
(25)	int/175.vpr, int/255.vortex
(26)	int/181.mcf, fp/168.wupwise
(27)	int/181.mcf, int/256.bzip2
(28)	int/186.crafty, int/164.gzip
(29)	fp/172.mgrid, fp/301.apsi
(30)	fp/177.mesa, fp/179.art
(31)	fp/200.sixtrack, fp/183.equake
(32)	int/254.gap, int/197.parser
(33)	int/186.crafty, int/300.twolf
(34)	int/300.twolf, int/254.gap
(35)	fp/301.apsi, fp/188.ammp
(36)	int/254.gap, int/175.vpr
(37)	fp/301.apsi, int/175.vpr
(38)	int/181.mcf, fp/301.apsi
(39)	int/164.gzip, fp/301.apsi

Table 7.2: Benchmark mix in 2-Thread configuration - SPEC CPU2000 suite. First column is the workload mix number and the second column lists the benchmarks executed concurrently as part of the multiprogrammed workload.

(0)	fp/179.art, int/256.bzip2 fp/173.applu, fp/188.ammp
(1)	fp/168.wupwise, int/181.mcf fp/188.ammp, int/255.vortex
(2)	fp/179.art, int/300.twolf int/181.mcf, int/197.parser
(3)	int/186.crafty, int/181.mcf int/175.vpr, fp/168.wupwise
(4)	fp/188.ammp, int/254.gap int/164.gzip, int/175.vpr
(5)	fp/177.mesa, int/175.vpr int/164.gzip, int/164.gzip
(6)	fp/177.mesa, int/181.mcf fp/172.mgrid, int/186.crafty
(7)	int/181.mcf, int/164.gzip int/186.crafty, int/300.twolf
(8)	fp/200.sixtrack, fp/177.mesa fp/200.sixtrack, fp/188.ammp
(9)	fp/171.swim, int/186.crafty fp/200.sixtrack, fp/171.swim
(10)	fp/168.wupwise, fp/168.wupwise int/181.mcf, fp/177.mesa
(11)	fp/301.apsi, int/255.vortex int/255.vortex, fp/183.equake
(12)	int/254.gap, fp/173.applu fp/301.apsi, fp/173.applu
(13)	fp/200.sixtrack, fp/200.sixtrack int/197.parser, fp/171.swim
(14)	fp/171.swim, fp/301.apsi int/181.mcf, fp/177.mesa
(15)	int/181.mcf, fp/183.equake fp/301.apsi, fp/177.mesa
(16)	int/181.mcf, int/254.gap int/254.gap, int/255.vortex
(17)	fp/177.mesa, int/181.mcf int/300.twolf, fp/301.apsi
(18)	int/181.mcf, fp/183.equake int/254.gap, fp/200.sixtrack
(19)	fp/301.apsi, fp/179.art int/300.twolf, fp/200.sixtrack

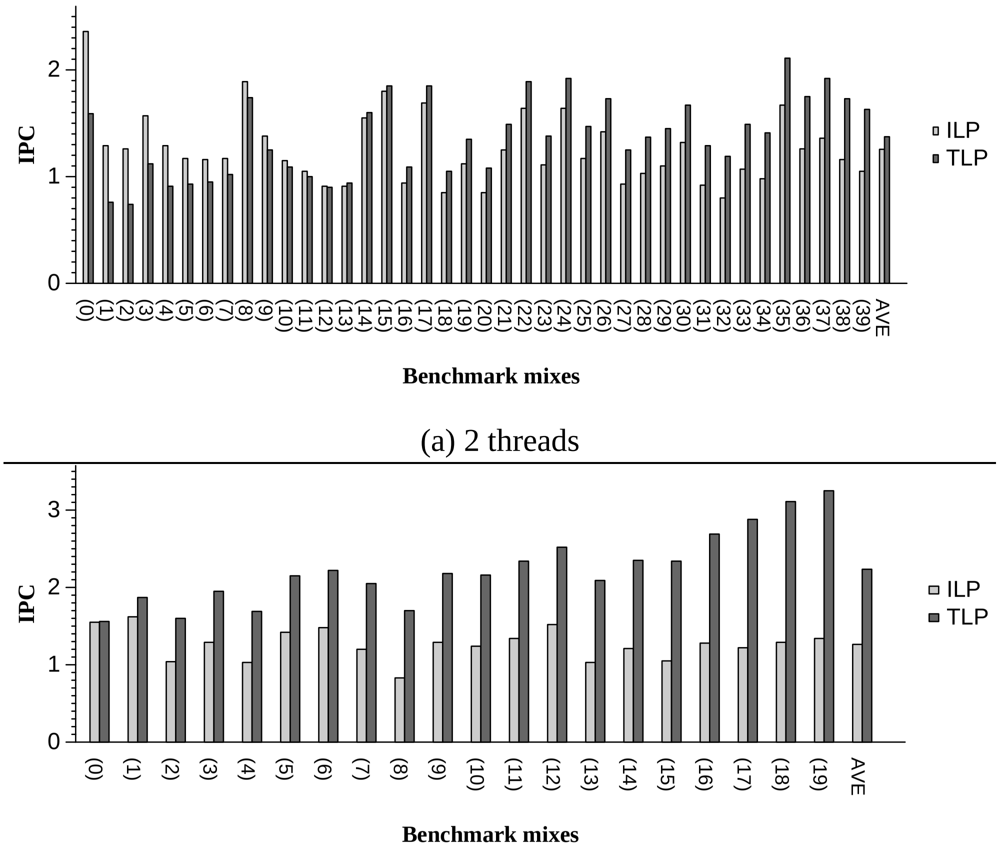
Table 7.3: Benchmark mix in 4-Thread configuration - SPEC CPU2000 suite. First column is the workload mix number and the second column lists the benchmarks executed concurrently as part of the multiprogrammed workload.

7.2.2 EEMBC Benchmarks

The EEMBC benchmarks results are very similar to the results for the SPEC CPU2000 benchmarks. We briefly summarize the results and our observations below. Figure 7.4 shows the IPC comparison between ILP and TLP-mode while running two threads and four threads. Mixes 13 through 39 in 2-Thread configuration, and all 20 mixes in the 4-Thread configuration, show higher utilization in the TLP-mode than the ILP execution of the program. The IPCs in are range of 0.74 to 2.16, with an average of 1.4 for the 2-Thread configuration, and range from 1.56 to 3.25, with an average of 2.2 in the 4-Thread configuration. The TLP-mode performance is better on the EEMBC benchmarks because they have limited parallelism, and therefore the potential for performance increase when increasing processor resources is less. In fact, as shown in Table B.2, the performance losses when reducing the instruction window are lower for the EEMBC benchmarks than the SPEC CPU2000 benchmarks. Recall that one of the primary effects of multi-threading is the reduced instruction window size each program sees.

Figure 7.5 shows the speedup achieved in the TLP-mode compared to the ILP-mode. More than half of the 40 mixes in the 2-Thread configuration (28) show a speedup, on average 10%, while all the 20 mixes show a speedup in the 4-Thread configuration, on average 80%. The speedups achieved in the EEMBC benchmarks are less than the speedups achieved with the SPEC CPU200 benchmarks, which have more parallelism.

The efficiency of the TLP-mode is slightly higher on the EEMBC benchmarks compared to the SPEC CPU2000 benchmarks. The average *max* efficiency

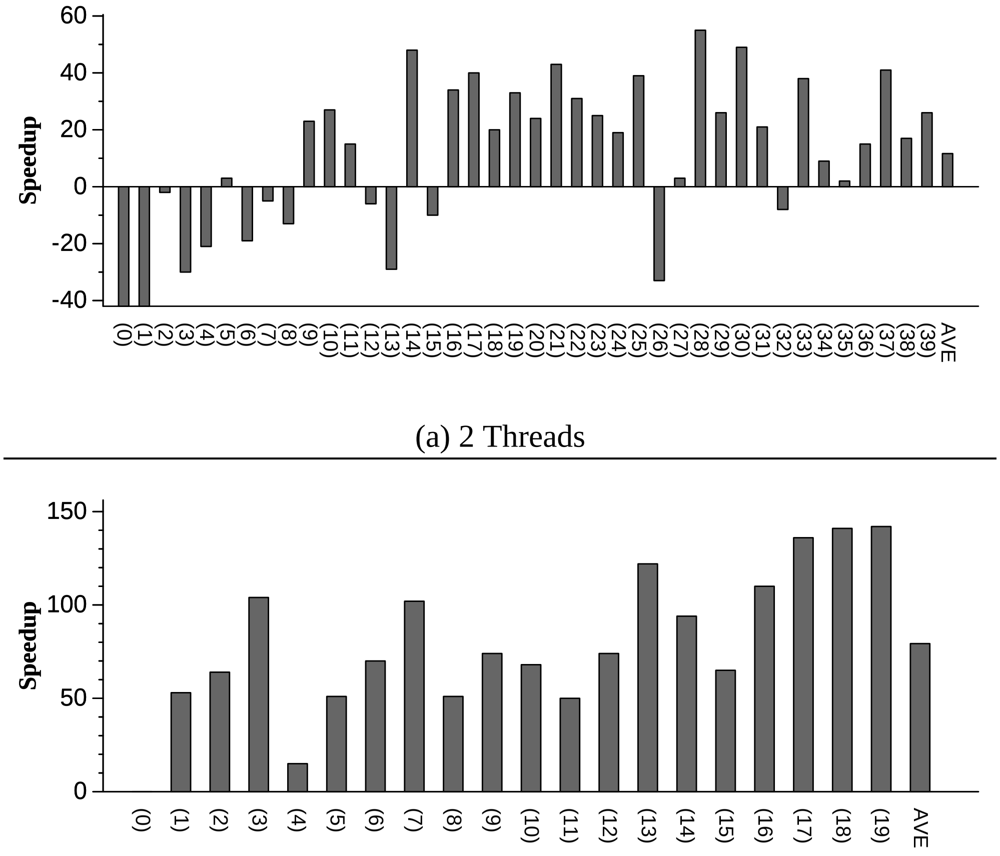


(a) 2 threads

(b) 4 threads

Figure 7.4: TLP-mode performance (utilization) - EEMBC suite.

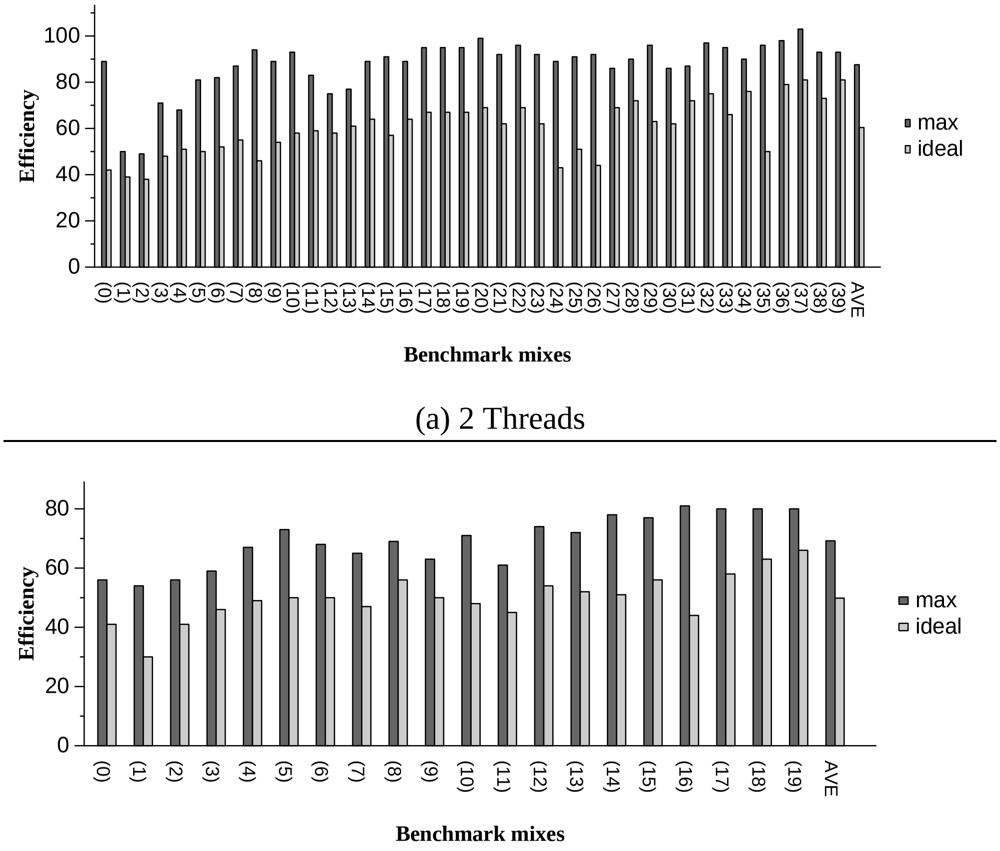
is 87% for the 2-Thread configuration and is 70% on the 4-Thread configuration. The *ideal* efficiency is 60% on the 2-Thread configuration and 50% on the 4-Thread configuration.



(a) 2 Threads

(b) 4 Threads

Figure 7.5: TLP-mode speedup compared to serialized execution - EEMBC suite.



(a) 2 Threads

(b) 4 Threads

Figure 7.6: TLP-mode execution efficiency - EEMBC suite.

(0)	automotive/rspeed01, telecom/fft00
(1)	automotive/aifftr01, automotive/aifirf01
(2)	automotive/aifirf01, automotive/aiifft01
(3)	automotive/iirflt01, consumer/cjpeg
(4)	automotive/a2time01, consumer/djpeg
(5)	office/bezier02, automotive/aifirf01
(6)	office/bezier02, automotive/tblook01
(7)	automotive/a2time01, automotive/bitmnp01
(8)	automotive/puwmod01, telecom/fft00
(9)	automotive/iirflt01, telecom/fbital00
(10)	automotive/bitmnp01, telecom/autocor00
(11)	automotive/tblook01, office/text01
(12)	automotive/basefp01, networking/routelookup
(13)	networking/routelookup, automotive/tblook01
(14)	consumer/cjpeg, automotive/puwmod01
(15)	office/dither01, automotive/idctrn01
(16)	networking/ospf, automotive/iirflt01
(17)	automotive/canrdr01, office/dither01
(18)	automotive/aifirf01, automotive/rspeed01
(19)	automotive/ttsprk01, networking/pktflow
(20)	automotive/rspeed01, automotive/basefp01
(21)	automotive/pntrch01, telecom/fbital00
(22)	automotive/puwmod01, office/dither01
(23)	networking/pktflow, office/text01
(24)	consumer/cjpeg, telecom/fft00
(25)	office/text01, telecom/conven00
(26)	telecom/fft00, office/text01
(27)	networking/ospf, automotive/pntrch01
(28)	automotive/pntrch01, office/text01
(29)	automotive/rspeed01, automotive/idctrn01
(30)	automotive/aifftr01, automotive/bitmnp01
(31)	automotive/ttsprk01, networking/routelookup
(32)	automotive/ttsprk01, automotive/iirflt01
(33)	automotive/bitmnp01, automotive/canrdr01
(34)	automotive/ttsprk01, automotive/matrix01
(35)	telecom/fft00, automotive/aiifft01
(36)	consumer/djpeg, networking/routelookup
(37)	automotive/rspeed01, office/rotate01
(38)	automotive/aiifft01, office/text01
(39)	telecom/viterb00, automotive/pntrch01

Table 7.4: Benchmark mix in 2-Thread configuration - EEMBC suite. First column is the workload mix number and the second column lists the benchmarks executed concurrently as part of the multiprogrammed workload.

(0)	automotive/a2time01, consumer/cjpeg automotive/ttsprk01, automotive/ttsprk01
(1)	telecom/fft00, automotive/aiifft01 automotive/tblook01, automotive/idctrn01
(2)	telecom/viterb00, automotive/a2time01 networking/routelookup, office/bezier02
(3)	consumer/djpeg, consumer/djpeg automotive/puwmod01, automotive/iirflt01
(4)	telecom/viterb00, automotive/basefp01 networking/ospf, automotive/tblook01
(5)	office/dither01, automotive/cacheb01 automotive/aifirf01, networking/pktflow
(6)	consumer/cjpeg, automotive/matrix01 automotive/rspeed01, automotive/rspeed01
(7)	office/rotate01, office/text01 automotive/a2time01, automotive/bitmnp01
(8)	automotive/aifirf01, automotive/puwmod01 automotive/ttsprk01, automotive/cacheb01
(9)	consumer/djpeg, automotive/pntrch01 automotive/rspeed01, consumer/djpeg
(10)	office/rotate01, networking/pktflow automotive/basefp01, office/bezier02
(11)	automotive/iirflt01, automotive/aifftr01 consumer/djpeg, office/dither01
(12)	automotive/pntrch01, automotive/puwmod01 consumer/cjpeg, automotive/bitmnp01
(13)	office/text01, automotive/pntrch01 automotive/iirflt01, automotive/idctrn01
(14)	automotive/canrdr01, office/bezier02 telecom/fbital00, automotive/ttsprk01
(15)	automotive/bitmnp01, automotive/canrdr01 office/text01, automotive/ttsprk01
(16)	telecom/conven00, office/text01 telecom/fbital00, telecom/fbital00
(17)	automotive/rspeed01, automotive/matrix01 office/rotate01, telecom/fbital00
(18)	telecom/viterb00, office/rotate01 consumer/djpeg, networking/ospf
(19)	telecom/viterb00, office/rotate01 automotive/ttsprk01, office/rotate01

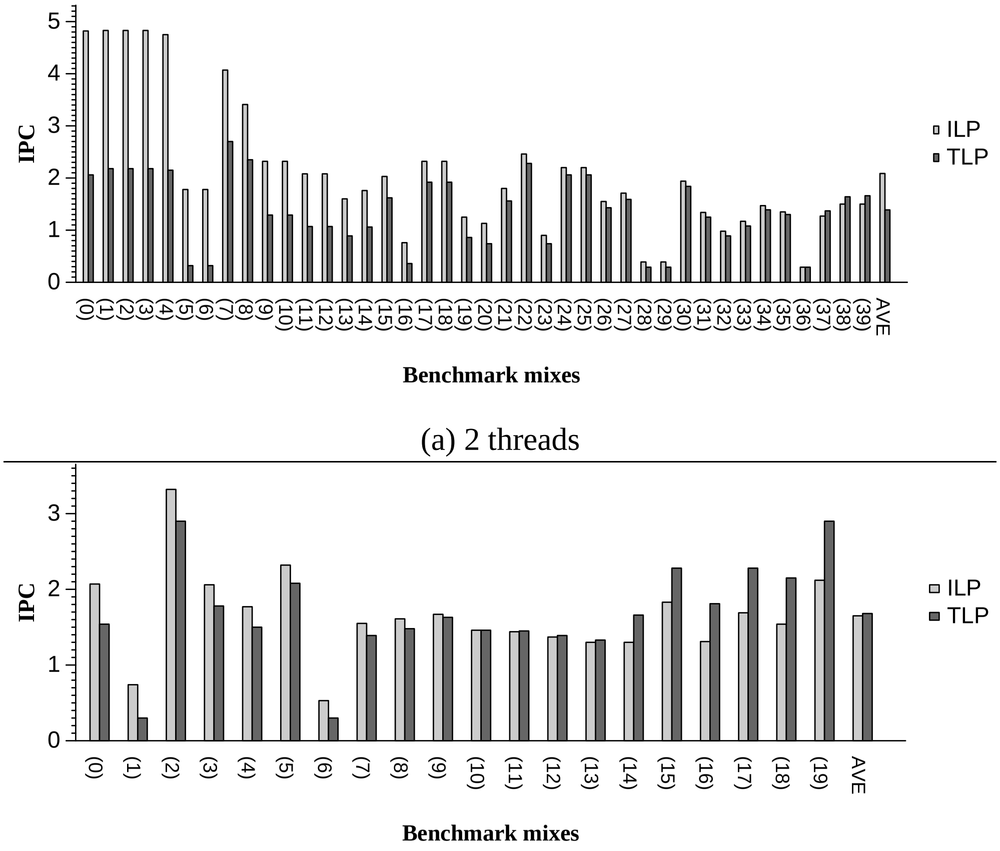
Table 7.5: Benchmark mix in 4-Thread configuration - EEMBC suite. First column is the workload mix number and the second column lists the benchmarks executed concurrently as part of the multiprogrammed workload.

7.2.3 Data Parallel Benchmarks

Overall the data parallel benchmark kernels benefit very little from running in TLP-mode. Figure 7.7 shows the IPC comparison of the TLP-mode and ILP-mode for the data parallel benchmarks. Overall only 4 of 40 program mixes show high processor utilization while running in the 2-Thread configuration and by only 6% better on average, and 10 of 20 program mixes perform better while running in the 4-Thread TLP-mode, and only by 15% better on average. All of the data parallel benchmarks have abundant parallelism in them, and executing them in TLP-mode introduces a lot of contention between the programs for shared resources like the data cache, operand network, and the register files. Furthermore, they show a significant slowdown when they are executed with reduced resources of 256 reservation stations compared to 1024 reservation stations. As a result, the 2-Thread configuration which leaves half the processor's reservation stations un-utilized performs quite poorly. The 4-Thread configuration perform slightly better, but still not as well as executing a single thread.

Figure 7.8 shows speedup achieved by TLP-mode execution compared to ILP-mode serial execution. For the 2-Thread configuration, since the utilization is poorer in the TLP-mode, it is natural to expect poor speedups. In fact, on average there is a 27% slowdown, and the best case speedup is only 10%. The 4-Thread configuration is slightly better, on average it performs identical to the ILP-mode. Best case speedup is 39% and in the worst case, slowdown is 60%. Since these programs have abundant parallelism coupled with many memory accesses, executing multiple of them in parallel causes a lot of contention for shared resources and thereby hinders TLP-mode execution.

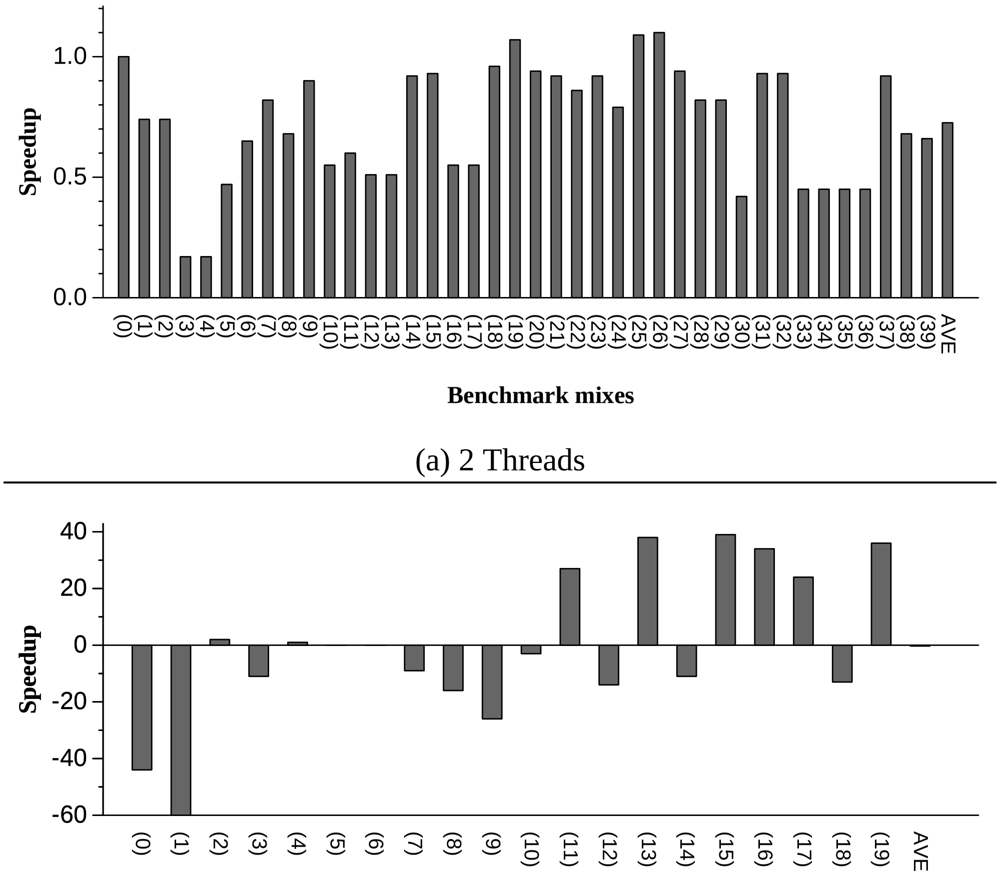
The efficiency of the TLP-mode is much lower in the DLP suite compared to the both the SPEC CPU2000 and the EEMBC suites. In the 2-Thread configuration, on average the *max* efficiency is only 63% and is as slow as 13%. In the 4-Thread configuration, the *max* efficiency is even worse, with an average of only 40%. The *ideal* efficiency is even worse and is 33% and 19% on average for the 2-Thread and 4-Thread configuration. Since the DLP programs have ample parallelism, when executed in isolation they can very effectively use the parallelism and concurrent execution in TLP-mode introduce a lot of contention. These results suggests that for the DLP programs, the contention overheads in the TLP-mode are quite significant, and secondly that TLP execution in general is not a very efficient use of processor resources for these benchmarks.



(a) 2 threads

(b) 4 threads

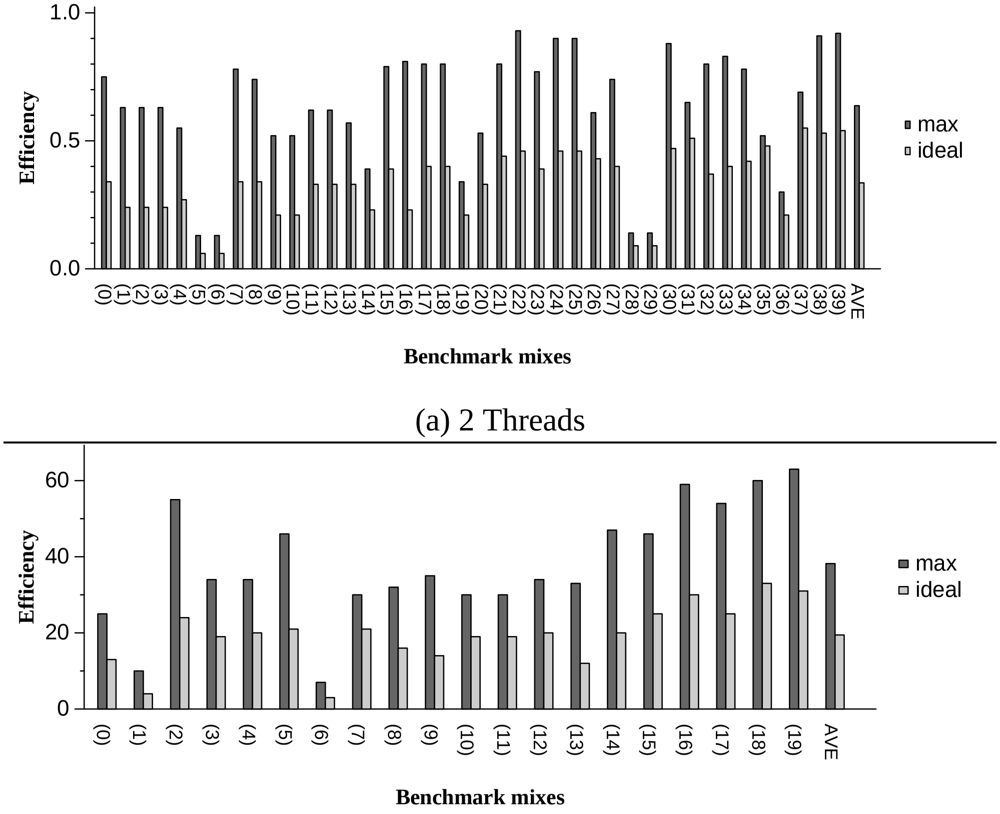
Figure 7.7: TLP-mode performance (utilization) - DLP suite.



(a) 2 Threads

(b) 4 Threads

Figure 7.8: TLP-mode speedup compared to serialized execution - DLP suite.



(a) 2 Threads

(b) 4 Threads

Figure 7.9: TLP-mode execution efficiency - DLP suite.

(0)	scientific/fft, DSP/highpassfilter
(1)	DSP/highpassfilter, DSP/convert
(2)	DSP/highpassfilter, DSP/convert
(3)	DSP/convert, DSP/highpassfilter
(4)	DSP/highpassfilter, DSP/dct
(5)	network/rijndael, DSP/highpassfilter
(6)	DSP/highpassfilter, network/rijndael
(7)	graphics/vertexskinning, DSP/highpassfilter
(8)	DSP/dct, DSP/convert
(9)	graphics/vertexreflection, DSP/convert
(10)	graphics/vertexreflection, DSP/convert
(11)	scientific/fft, graphics/vertexreflection
(12)	scientific/fft, graphics/vertexreflection
(13)	scientific/fft, graphics/fragmentreflection
(14)	DSP/dct, graphics/fragmentreflection
(15)	graphics/fragmentsimplelight, graphics/vertexreflection
(16)	network/md5, network/md5
(17)	graphics/fragmentsimplelight, graphics/vertexskinning
(18)	graphics/fragmentsimplelight, graphics/vertexskinning
(19)	network/blowfish, DSP/dct
(20)	scientific/fft, network/blowfish
(21)	graphics/vertexsimplelight, graphics/fragmentreflection
(22)	graphics/vertexreflection, graphics/vertexskinning
(23)	network/blowfish, network/md5
(24)	graphics/fragmentreflection, graphics/vertexskinning
(25)	graphics/fragmentreflection, graphics/vertexskinning
(26)	scientific/LU, graphics/vertexsimplelight
(27)	graphics/vertexskinning, network/blowfish
(28)	DSP/dct, network/rijndael
(29)	network/rijndael, DSP/dct
(30)	graphics/fragmentsimplelight, graphics/vertexsimplelight
(31)	scientific/LU, scientific/fft
(32)	network/md5, graphics/fragmentreflection
(33)	graphics/vertexsimplelight, network/md5
(34)	network/blowfish, graphics/vertexreflection
(35)	scientific/LU, scientific/LU
(36)	network/rijndael, scientific/fft
(37)	scientific/LU, network/blowfish
(38)	graphics/vertexsimplelight, network/blowfish
(39)	network/blowfish, graphics/vertexsimplelight

Table 7.6: Benchmark mix in 2-Thread configuration - DLP suite. First column is the workload mix number and the second column lists the benchmarks executed concurrently as part of the multiprogrammed workload.

(0)	scientific/LU, DSP/highpassfilter scientific/LU, DSP/convert
(1)	graphics/fragmentreflection, network/rijndael DSP/convert, network/md5
(2)	DSP/highpassfilter, graphics/vertexsimplelight DSP/convert, scientific/fft
(3)	DSP/highpassfilter, network/blowfish scientific/LU, graphics/vertexsimplelight
(4)	scientific/LU, scientific/fft graphics/vertexskinning, graphics/fragmentsimplelight
(5)	DSP/convert, scientific/fft graphics/vertexskinning, graphics/vertexsimplelight
(6)	scientific/fft, DSP/convert network/rijndael, DSP/dct
(7)	scientific/fft, scientific/LU graphics/vertexskinning, scientific/LU
(8)	scientific/LU, graphics/fragmentreflection graphics/vertexreflection, DSP/convert
(9)	network/blowfish, graphics/vertexreflection DSP/convert, DSP/convert
(10)	scientific/LU, graphics/vertexsimplelight DSP/dct, network/blowfish
(11)	graphics/vertexreflection, DSP/dct scientific/LU, network/blowfish
(12)	scientific/fft, DSP/dct network/blowfish, graphics/fragmentreflection
(13)	DSP/convert, network/md5 graphics/fragmentsimplelight, DSP/convert
(14)	graphics/fragmentreflection, DSP/convert graphics/fragmentsimplelight, network/md5
(15)	graphics/vertexskinning, network/blowfish DSP/dct, graphics/fragmentsimplelight
(16)	graphics/fragmentreflection, graphics/vertexsimplelight graphics/fragmentreflection, network/md5
(17)	graphics/vertexsimplelight, DSP/convert graphics/fragmentsimplelight, network/blowfish
(18)	graphics/fragmentreflection, graphics/fragmentreflection graphics/vertexreflection, network/blowfish
(19)	DSP/highpassfilter, network/blowfish graphics/vertexsimplelight, network/blowfish

Table 7.7: Benchmark mix in 4-Thread configuration - DLP suite. First column is the workload mix number and the second column lists the benchmarks executed concurrently as part of the multiprogrammed workload.

7.3 Summary

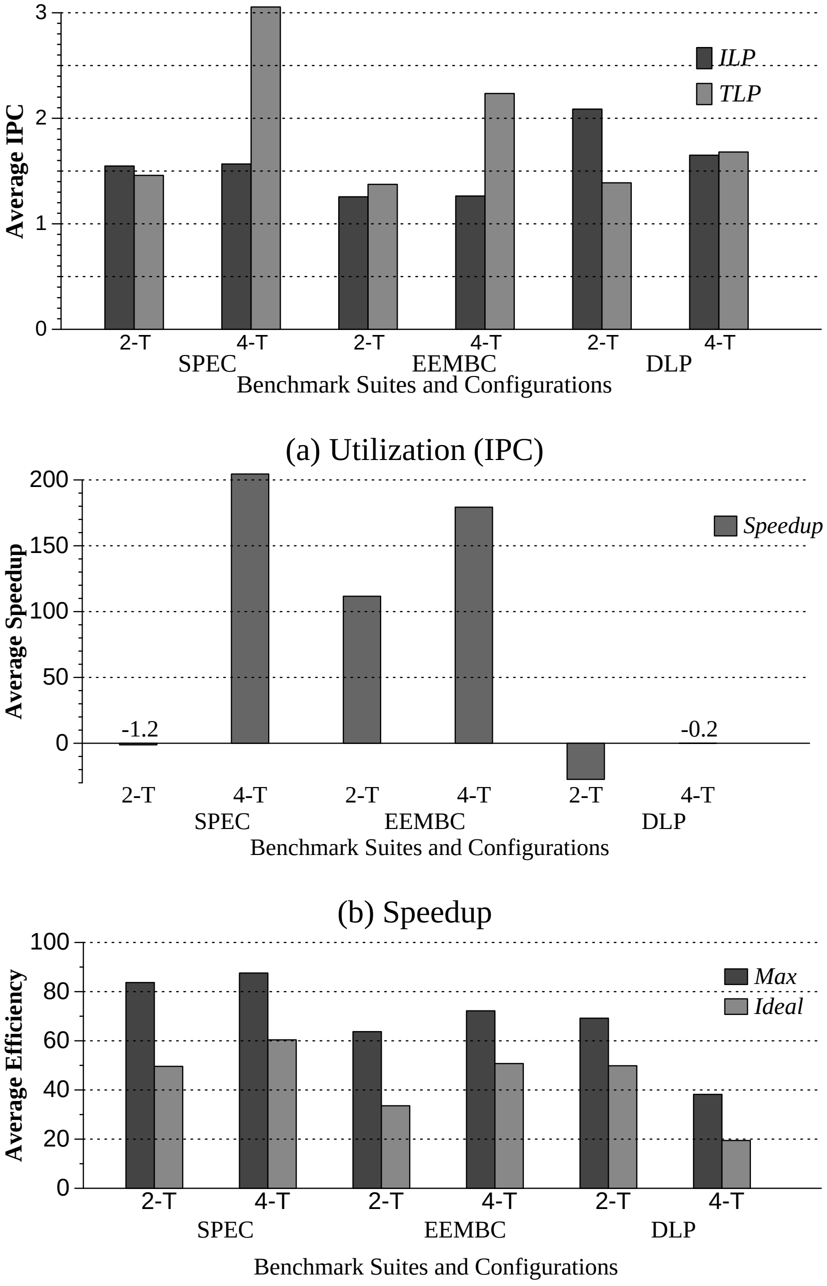
Overall, the TLP-mode is quite effective at utilizing the processor resources to execute multi-programmed workloads. The polymorphous mechanisms provide an execution window with reduced speculation depth for each processor, and a memory system and register file with less apparent bandwidth for each program compared to the ILP-mode of the processor. We studied the performance of the TLP-mode on three benchmark suites: SPEC CPU200, EEMBC, and our DLP suite, with randomly generated program mixes. Figure 7.10 shows the average processor utilization (IPC), speedup, and efficiency across the three benchmark suites. We observed that the random generation of program mixes creates significantly diverse program behavior. The diversity of the workloads, control speculation, and resource contention most significantly influence TLP-mode performance.

Workload: The processor utilization, speedup, and efficiency are significantly affected by the workload. While, the SPEC CPU2000 workload mixes show an IPC of 3.2 in the 4-Thread TLP-mode, the DLP workloads sustain only 1.6. The SPEC CPU2000 and the DLP suites show almost opposite behavior, with the EEMBC suite being in-between. The SPEC CPU2000 benchmarks show the highest speedup (close to 200%) and efficiency (60%), while the speedup is slightly less than 1% and efficiency of the DLP benchmarks is only 20% in the 4-Thread mode. The poor performance of the DLP workloads is primarily because of the ample parallelism and large amount of memory accesses in them, which causes a lot of contention losses in TLP-mode execution.

Control speculation: By using multithreading, the processor is able to effectively generate useful work and is often significantly better than using control speculation to generate useful work from a single thread. In fact, the reduced speculation depth helps tremendously in programs that have poor control speculation behavior, coupled with small block sizes and limited parallelism. By executing multiple threads, the processor resources are used to extract parallelism from different threads. This effect is most dramatic in the SPEC CPU2000 and EEMBC benchmarks.

Contention: The primary hindrance to performance that we expected was resource contention for the shared resources between the threads. We found that while the resource contention did grow significantly, only in the case of programs with large amounts of parallelism did it affect performance. We measured the resource contention for the critical processor resources like the data cache ports, operand network, and the register files. Table 7.8 lists the percentage of cycles that the execution tiles are stalled due to a resource conflict at any of these structures in the processor.

The second column shows the resource contention in ILP-mode, and the third and fourth columns show resource contention in TLP-mode in the 2-Thread and 4-Thread configuration. Between the 2-Thread configuration and the ILP-mode cycles lost due to contention drops because half of the reservations stations are unused and the processor is in general under-utilized. Comparing resource contention between the ILP-mode and the 4-Thread TLP-mode, a significant increase is seen, with the largest increase seen in DLP benchmarks.



(a) Utilization (IPC)

(b) Speedup

(c) Efficiency

Figure 7.10: TLP-mode summary of results. FIXME change to %

Benchmark suite	ILP	2-Threads	4-Threads
SPEC CPU2000	20	19	14
EEMBC	10	11	21
Data parallel benchmarks	20	10	50

Table 7.8: Resource contention: percentage of cycles that the execution tiles are stalled due to a resource conflict.

Summary: The results demonstrate that the polymorphous mechanisms are effective at creating an illusion of a full processor for each program. In terms of implementation complexity the changes required are quite small, control logic changes in the instruction select logic, register renaming logic, and modifications to some table lookups in the branch predictor. Going against the spirit of polymorphism, adding TLP support requires addition of extra architectural register file storage for the different threads and a small amount of extra storage in the next-block predictor.

It will be interesting to evaluate in detail the scalability of the TLP-mode. Due to simulation constraints and constraints of the design, we evaluated a maximum of 4 threads executing. Studying how deeply this can be scaled is an interesting question to explore. Also in this study we did not measure the power consumption aspects of the TLP-mode. While the implications for power saving techniques like clock-gating are not drastically different from the ILP, the heuristics may need to be changed a little compared to the ILP mode. In this study, we did not evaluate true, multithreaded workloads with interacting threads. Studying the data sharing effects and resource constraints for these workloads is another interesting future direction to explore.

Chapter 8

Data-Level Parallelism

Data-level parallelism is typically characterized by independent operations applied to a large number of data records. Historically, systems targeted at DLP have been regular architectures like vector processors, systolic arrays, and SIMD arrays optimized for simple control and exploiting the regularity in the instruction stream and data stream. Such architectures had narrow application domains, but more recently hybrid SIMD-VLIW architectures like the Imagine architecture and multimedia ISA extensions have been targeted at DLP workloads and have provided more diversity.

The main focus of this chapter is a systematic analysis of DLP in the polymorphism context. We first perform a detailed analysis of DLP workloads by characterizing their fundamentals in terms of memory behavior, control behavior, and computation. We then quantitatively analyze the bottlenecks in conventional microarchitectures for DLP processing. Based on this analysis and the fundamental program behavior we determine a core set of polymorphous mechanisms to support data-level parallelism.

The remainder of this chapter is organized as follows. In Section 8.1 we motivate the need for a detailed analysis of DLP workloads and summarize the historical evolution and recent trends in data parallel architectures. In Section 8.2 we provide a detailed characterization of the fundamental behavior of DLP workloads and in Section 8.3 we evaluate these workloads using a conventional execution model to determine the bottlenecks that hinder DLP execution. In Section 8.4 we use the application characterization to develop a set of flexible microarchitecture mechanisms. Finally, in Section 8.5 we present performance results that can be obtained using these mechanisms and compare the results to specialized DLP architectures.

8.1 DLP Overview and History

Data-parallel programs are growing in importance, increasing in diversity, and demanding increased performance from hardware. Specialized hardware is commonplace in the real-time graphics, signal processing, network processing, and high-performance scientific computing domains. Modern graphics processors have rapidly evolved from 20 GFlops (at 450 MHz) in 2003 [27] to 360 GFlops (at 650 MHz) in the latest ATI Radeon R580, in late 2006. Based on these levels of performs we can conclude that the number of single precision floating point units has grown from approximately 40 to more than 500. Software radios for 3G wireless baseband receivers are being developed for digital signal processors and require 15 Gops to deliver adequate performance [131]. Each arithmetic processor in the Earth Simulator contains forty eight vector pipelines and delivers peak performance of up to 8 GFlops. The Cell processor in the Playstation3 system has a theoretical peak performance of 25.6 GFlops provided by each SIMD core called SPEs running at 3.2 GHz [84], and the Playstation3 system has been reported as being able to provide 2 TFlops. The Xbox360 system has an estimated peak performance of 1 TFlops [9]. While these domains of data-parallel applications have many common characteristics, they typically show differences in the types of memory accesses, computation requirements, and control behavior.

Most data-parallel architectures target a subset of data-parallel programs, and have poor support for applications outside of that subset. Vector architectures provide efficient execution for programs with mostly regular memory accesses and simple control behavior. However, the vector model is less effective on programs that require computation across multiple vector elements or access memory in an unstructured or irregular fashion. SIMD architectures provide support for communication between execution units (thereby enabling computation across multiple data elements), but are also globally synchronized and hence provide poor support for applications with conditional execution and data dependent branches. MIMD architectures have typically been constructed of coarse-grained processors and operate on larger chunks of data using the single-program, multiple data (SPMD) execution model, with poor support for fine-grained synchronization. Emerging applications, such as real-time graphics, exhibit control behavior that requires fine-grained MIMD execution and fine-grained communication among execution units.

Many data-parallel applications which consist of components that exhibit different characteristics are often implemented on specialized hardware units. For example, most real-time graphics processing systems use specialized hardware coupled with the programmable components for *MPEG4* decoding. The TMS320C6416

DSP chip integrates two specialized units targeted at convolution encoding and forward error correction processing. While many of these specialized accelerators have been dedicated to a single narrow function, architectures are now emerging that consist of multiple programmable data-parallel processors that are specialized in different ways. The Sony Emotion Engine included two specialized vector units– one tuned for geometry processing in graphics rendering and the other specialized for behavioral and physical simulation [101]. The Sony Handheld Engine integrates a DSP core, a 2D graphics core and an ARM RISC core on a single chip, each targeted at a distinct type of data-parallel computation.

Design Convergence: Integrating many such specialized DLP cores leads to increased design cost and area, since different types of processors must be designed and integrated together. While data-level parallelism is one fundamental property that affects the processor organization, DLP workloads are varied enough that a detailed analysis of these workloads is required to understand their behavior.

In this dissertation, we identify and characterize the application demands of different data parallel program classes. While these classes have some common attributes, namely high computational intensity and high memory bandwidth, we show that they also have important differences in their memory access behavior, instruction control behavior and instruction storage requirements. As a result, different applications can demand different hardware capabilities varying from simple enhancements, like efficient lookup tables, to different execution models, such as SIMD or MIMD.

Based on the program attributes identified, we propose a set of polymorphous microarchitectural mechanisms for augmenting the execution core, instruction control, and memory system to build a flexible data-parallel architecture. The mechanisms are universal, since they support each type of DLP behavior and can be applied to diverse architectures ranging from vector processors to superscalar processors. In this dissertation we use the TRIPS architecture as a baseline for performance evaluation. We also show a rough comparison of the performance of these mechanisms to current best-of-breed specialized processors in each application domain.

Dataflow graph abstraction: The TRIPS processor is well suited for data-parallel execution with its high functional unit density, efficient ALU-ALU communication, high memory bandwidth, and technology scalability. The dataflow style ISA design provides several relevant capabilities, including the ability to map various communication patterns and statically placed dynamically issued execution, that enable a straight-forward implementation of the mechanisms. No major changes to the ISA or programming model is required. The partitioned design of the on-chip memory also is well suited for the bandwidth augmentations that we propose to address the high bandwidth requirement of these applications. Remaining true to the spirit of polymorphism, the DLP mechanisms largely modify only the control path to create flexible behavior without adding more datapath or storage elements.

8.2 Application Behavior

Data-parallel workloads can be classified into domains based on the type of data being processed. The nature of computation varies within a domain and across the different domains. The applications vary from simple computations on image data converting one color space to another (comprising 10s of instructions), to complex encryption routines on network packets (comprising 100s of instructions). Four broad categories cover a significant part of this spectrum: digital signal processing, scientific, network/security, and real-time graphics. In this section, we first describe the behavior of these applications categorized by three parts of the architecture they affect: memory, instruction control, and execution core. We then describe our suite of data-parallel programs and present their attributes.

8.2.1 Program Attributes

At an abstract level, data-parallel programs consist of a loop body executing on different parts of the input data. In a data parallel architecture this loop body is typically executed on different execution units, operating on different parts of memory in parallel. We refer to this loop body as a *kernel*. Typically the iterations of a loop are independent of each other and can execute concurrently. Kernels exhibit different types of memory accesses and control behavior, as well as varying computation needs. One example of data-parallel execution is the computation of a 2D discrete cosine transform (DCT) on 8x8 blocks of an image. In this case, parallelism can be exploited by processing the different 8x8 blocks of the image on different computation nodes concurrently. The processing of each instance of the kernel is identical and can be performed in a globally synchronous manner across different computation nodes. A more complex data-parallel computation is a technique called *skinning* which is used for animation in graphics processing. A dynamically varying number of matrix-vector multiplies are performed at each polygon vertex in a 3D model. The different vertices in the model can be operated upon in parallel, completely independent of each other, but the amount of computation varies from vertex to vertex.

Memory behavior: The memory behavior of data-parallel applications can be classified into four different types: (1) regular memory accesses, (2) irregular memory accesses, (3) named constant scalar operands, and (4) indexed constant operands. In characterizing DLP programs, we are interested in the frequency of occurrence of each of the four types of accesses in a kernel. The four types of accesses are not exclusive and a kernel may make accesses from all four categories.

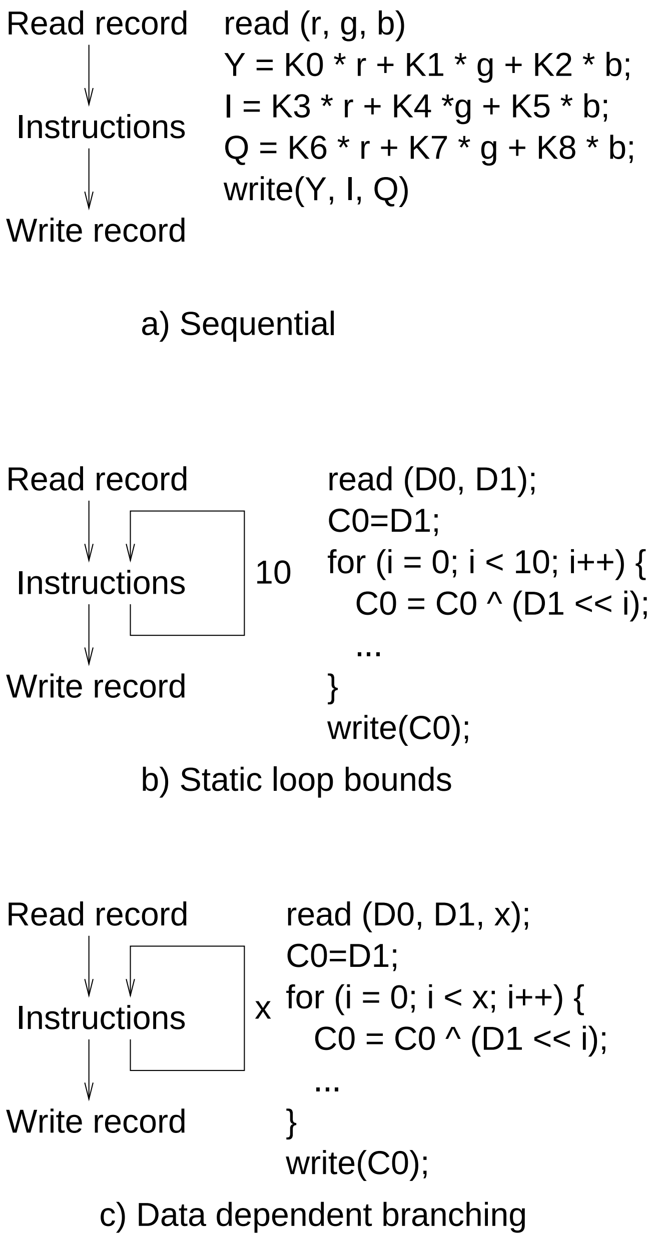
- *Regular memory:* Data-parallel kernels typically read from memory in a very structured manner (strided accesses for example). We use the term *record* to refer to a group of elements on which a single iteration of a kernel operates. In image processing, for example, a record may consist of 3 elements, corresponding to 3 primary color components. Because of the regularity of these accesses, microarchitectures can pipeline accesses or amortize the address calculation and other overheads associated with accessing memory, by issuing one instruction to fetch one or more full records.

- *Irregular memory:* Some data-parallel kernels access some parts of memory in a random access fashion similar to conventional sequential programs. One example of such behavior is texture accesses in graphics programs. Unlike regular memory accesses, the overheads of these accesses cannot be amortized by aggregating them and these accesses are not pre-computable before their use. Typical texture data structures for graphics scenes require several megabytes of storage.

- *Scalar constants:* Many operations in data parallel kernels use runtime constants that are unmodified through the full execution of the kernel, such as the constants used in convolution filters applied to an image. The number of coefficients is often small and can typically be stored in machine registers rather than memory.

- *Indexed constants:* Many DLP applications require small lookup tables with the index determined at runtime. Encryption kernels use such lookup tables with between 256 and 1024 8-bit entries to substitute one byte for another byte during computation. These accesses can be frequent in some kernels, reducing performance if they have long access latencies. Storing these tables in the level-1 data caches consumes little storage space, but tremendous cache bandwidth.

Control behavior: The complexity of the control structure in the kernel determines the type of synchronization and instruction sequencing required. Figure 8.1



```
read (r, g, b)
Y = K0 * r + K1 * g + K2 * b;
I = K3 * r + K4 *g + K5 * b;
Q = K6 * r + K7 * g + K8 * b;
write(Y, I, Q)
```

a) Sequential

```
read (D0, D1);
C0=D1;
for (i = 0; i < 10; i++) {
   C0 = C0 ^ (D1 << i);
   ...
}
write(C0);
```

b) Static loop bounds

```
read (D0, D1, x);
C0=D1;
for (i = 0; i < x; i++) {
   C0 = C0 ^ (D1 << i);
   ...
}
write(C0);
```

c) Data dependent branching

Figure 8.1: Kernel control behavior.

shows the three different types of control behavior possible.

- *Sequential instructions:* The simplest kernels contain a sequence of instructions with no internal control flow. A degenerate case is a single vector operation, but the 2D DCT can be transformed into this model by unrolling all of the internal computations of the 8x8 kernel. Each iteration of these kernels executes in the exact same fashion, so these kernels are well-suited for vector or SIMD control. Figure 8.1a shows this type of control behavior with example RGB to YIQ color conversion kernel pseudo-code.

- *Simple static loops:* A slightly more complex type of control behavior occurs when the kernel contains loops with static loop bounds. Figure 8.1b shows this type of control behavior with an example encryption kernel pseudo-code. Like the simple instruction sequences, each iteration of the kernel is the same and can be executed in a vector or SIMD style. Such kernels can be unrolled at compile time increasing the code size of the kernel, although for some kernels this transformation results in prohibitively large instruction storage requirements. Architectures that lack any branching support (like some graphics fragment processors) must rely on complete unrolling to execute such loops.

- *Runtime loop bounds:* Figure 8.1c shows the most generic of control behavior: data dependent branching. Such kernels would require masking instructions to execute on vector and SIMD machines, and are ideally suited to fine-grained MIMD machines, since each processing element can be independently controlled according to the local branching behavior.

Runtime conditionals, such as simple and nested `if-then-else` statements, can make any of these loop control templates more complex. Data-parallel architectures have traditionally implemented conditionals by using predication [118, 22], conditional streams [85], or vector masks [149]. Finer partitioning of control, such as provided by a fine-grained MIMD architecture can reduce or eliminate these overheads that conditionals have in highly synchronized architectures.

8.2.2 Benchmark Attributes

Table 8.1 describes a suite of DLP kernels selected from four major application domains. This the DLP suite used in our ILP study in chapter 6 and the TLP study in chapter 7. Tables 8.2 and 8.3 characterize these kernels according to the computation, memory and control criteria presented previously. The two computation columns list the number of instructions and inherent ILP within the kernel (ILP is the number of instructions in one iteration of a kernel, divided by the dataflow graph height; when the loop bound was variable, the kernel was completely unrolled). The first memory column lists the size of the record (in 64-bit words) that each kernel reads and writes, the second column gives the number of irregular memory accesses, and the third and fourth memory columns describe the use of static coefficients within the kernel and the size of the lookup table for indexed constants, if one is needed. The control column indicates the number of loop iterations within the kernel (if any) and whether the loop bounds are variable across kernel instances,

Benchmark	**Description**
Multimedia processing	
convert	RGB to YIQ conversion.
dct	A 2D DCT of an 8x8 image block.
highpassfilter	A 2D high pass filter.
Network processing, security (1500 byte packets)	
MD5	MD5 checksum.
Rijndael	Rijndael (AES) packet encryption.
Blowfish	Blowfish packet encryption.
Scientific codes	
FFT	1024-point complex FFT.
LU Decomposition	LU decomposition of a dense 1024x1024 matrix.
Real-time graphics processing. See [51].	
vertex-simple	Basic vertex lighting with ambient, diffuse, specular and emissive lighting.
fragment-simple	Basic fragment lighting with ambient, diffuse, specular and emissive lighting.
vertex-reflection	Vertex shader for a reflective surface.
fragment-reflection	Fragment shader rendering a reflective surface using cube maps.
vertex-skinning	A vertex shader used for animation with multiple transformation matrices.
anisotropic-filtering	A fragment shader implementing anisotropic texture filtering [126].

Table 8.1: Benchmark description.

	Computation		**Control**
Benchmark	**# Inst**	**ILP**	
convert	15	5	-
dct	1728	6	16
highpassfilter	17	3.4	-
fft	10	3.3	-
lu	2	1	-
md5	680	1.63	-
blowfish	364	1.98	16
rijndael	650	11.8	10
vertex-simple	95	4.3	-
fragment-simple	64	2.96	-
vertex-reflection	94	7.1	-
fragment-reflection	98	6.2	-
vertex-skinning	112	6.8	Variable
anisotropic-filter	80	2.1	Variable

Table 8.2: Benchmark Attributes.

	Memory			
Benchmark	**Record size (words) read/write**	**# Irregular memory accesses**	**# Constants**	**# Indexed scalar constants**
convert	3/3	-	9	-
dct	64/64	-	10	-
highpassfilter	9/1	-	9	-
fft	6/4	-	0	-
lu	2/1	-	0	-
md5	10/2	-	65	-
blowfish	1/1	-	2	256
rijndael	2/2	-	18	1024
vertex-simple	7/6	-	32	-
fragment-simple	8/4	4	16	-
vertex-reflection	9/2	-	35	-
fragment-reflection	5/3	4	7	-
vertex-skinning	16/9	-	32	288
anisotropic-filter	9/1	≤ 50	6	128

Table 8.3: Benchmark attributes.

in which case the kernels exhibit data dependent control and prefer a fine-grained MIMD execution model. In the *anisotropic-filter* kernel, for example, the number of instructions executed varies from about 150 to 1000 for each instance. In vector or SIMD architectures, which lack support for fine-grained branching, each instance would execute all 1000 instructions, using predication or other techniques for nullifying unwanted instructions.

Collectively, the benchmarks exhibit wide variation in each of the attributes, demonstrating diversity in the fundamental behavior of DLP applications. Based on examination, we found these common characteristics across the workloads. While this does not cover the all possible program behavior, what we have is an important subset. We used this application study to drive an identification of attributes and complementary microarchitectural mechanisms.

8.3 Microarchitecture Analysis

In the previous section we described the basic attributes of DLP programs. In this section we present a quantitative characterization of processor bottlenecks for data-level parallelism. In the next section we map these processor bottlenecks back to program behavior and derive a set of polymorphous mechanisms for data-level parallelism. This principled approach based on program behavior and processor bottleneck analysis provides wider application coverage and more flexibility to the resulting architecture than simply creating mechanisms to configure the processor like other architectures–SIMD array or vector processor, for example.

8.3.1 Methodology

We compile the applications coded using a sequential programming model and compiled using the TRIPS compiler to create TRIPS binaries. We simulate these binaries on TRIPS simulator and use *tsim-critical*, which can quantify different microarchitecture events that contribute to a program's critical path, to identify bottlenecks. We modeled a perfect L2 cache to minimize the memory system effects and isolate the processor bottlenecks. *tsim-critical* can also determine the maximum speedup possible given the processor resources and compiler, by removing all overhead microarchitecture events from the critical path and recomputing the critical path. We track three groups of microarchitecture events which are related to the three classes of mechanisms: fetch which is related to processor control, register accesses which is related to the execution core and data storage, and memory accesses which is related to data storage.

Fetch: All the block sequencing/prediction, fetch, and deallocate events are grouped together under this heading. For DLP workloads, since large repetitive execution is common, optimized block sequencing logic can significantly reduce the overhead introduced by many of these events.

Register accesses: All accesses to registers are included in this group: reading, writing, register renaming, delays to route operands from the register files to a consumer, and the delays to route block outputs to the register file. We analyze register accesses as a separate category because DLP programs often access the register files repeatedly to read runtime constants. Since this is a read only access, it provides an opportunity for optimization, since the register tiles are designed for the common case of the same register being read and written across blocks.

Memory accesses: All the microarchitecture events that contribute to store delays and load to use delays, which include cache access delays in the data tiles, delays to route addresses and values to the data tiles, and delays to route values back to consumers for loads. In this quantitative analysis we do not classify the memory access into the four categories presented in Section 8.2. Classifying memory accesses into one of the four types requires sophisticated compiler analysis that can determine run time constants and data structure analysis. In addition this must be coupled with the critical path analysis.

8.3.2 Analysis

Table 8.4 shows the percentage of the critical path that is spent in each of the three main groups of events. The second, third, and fourth columns show the contribution to the critical path from fetch, register file accesses, and memory accesses, and the last column shows maximum speedup possible on the TRIPS architecture if all microarchitecture overheads are removed. The number within parenthesis in the fourth column, shows the percentage of operand network critical cycles spent in routing operands and addresses from and to the data caches.

Fetch: Column two shows that on average, the instruction fetch related events account for close to 30% of the program cycles. For programs like *rijndael*, where

Benchmark	Percentage contribution			Speedup
	Fetch	Register access	Memory access	
DSP/convert	36.5	4.7	37.0 (38.71)	14.9
DSP/dct	40.9	4.2	33.9 (19.99)	11.9
DSP/highpassfilter	19.4	15.7	30.3 (23.54)	5.6
graphics/fragmentreflection	12.0	10.4	13.1 (11.55)	2.5
graphics/fragmentsimplelight	20.1	10.4	26.4 (19.21)	4.4
graphics/vertexreflection	13.1	13.8	32.4 (20.7)	5.4
graphics/vertexsimplelight	17.0	13.5	22.6 (16.82)	4.3
graphics/vertexskinning	25.8	0.7	63.1 (65.96)	7.6
network/blowfish	2.1	33.4	19.9 (20.44)	3.8
network/md5	17.1	7.5	1.2 (3.39)	10.3
network/rijndael	95.2	0.2	0.9 (40.36)	21.3
scientific/fft	75.7	0.4	11.8 (43.19)	19.3
scientific/LU	6.5	0.1	88.9 (75.96)	34.7
Average	29.3	8.8	29.3	11.2

Table 8.4: Critical path analysis.

the compiler is able to produce only small blocks (6 instructions on average), more than 95% of the program cycles are devoted to managing instruction fetch. By examining the program source code and analyzing program behavior we determined that *rijndael* provides an opportunity for concurrency at a coarser granularity than what is visible in a 1024-entry instruction window. It processes streams of data concurrently, and this level of concurrency can be exploited by providing a very fine-grained MIMD execution substrate.

Register accesses: The average contribution of register accesses to the program execution is only 8.8%, but ranges from less than 1% to more than 35% as shown in the third column. As expected, programs with few operations on scalar constants see little of their critical path devoted to register accesses. For example *fft* and *LU* are dominated by memory accesses and their register access contributions are less than 1. Register accesses become a bottleneck for applications that use a large number of runtime constants, which are register allocated. As result the register renaming logic and the fanout to route the values to all consumers become limiting factors.

Memory accesses: Several programs are dominated by the number of cycles spent in memory. This delay includes the contention delays at the routers and the banks to reach the data tile cache banks, and router contention delays while routing replies back to the consumers, intrinsic cache access delays, TLB lookups and load-store conflict detection delays.

We can see a correlation between the number of memory accesses to instruction ratio presented in Table 8.3 and the fraction of critical cycles contributed to by memory accesses. *blowfish*, *rijndael*, *vertexskinning*, *fft*, and *LU* are all dominated by a large number of memory accesses. Recall that the compiler cannot register allocate indexed scalar constants and these result in memory accesses as well. Correspondingly the memory access contribution to the critical path varies from 40% to over 75%. Furthermore for programs with predominantly structured memory accesses like *fft* and *LU*, significant part of the operand network delays are spent in routing values to and from the memory system, as shown by the numbers within parenthesis in the fourth column. Speeding up these accesses can provide significant performance improvements.

Speedup: The last column in Table 8.4 shows the speedup that can be achieved if all microarchitecture overheads in the TRIPS processor are removed (the physical resources are still the same– 1024-wide instruction window, 16-wide issue, and 128 registers). We use a broad definition of *microarchitecture overheads*: all processor events, apart from the functional execution of an instruction, and the delays incurred as a result of these events is overhead. The speedup derived from this definition of overhead does not account for any potential changes to the software model or programming model.

The speedup values range from 2.5X to almost 35X, indicating there are significant microarchitecture overheads while executing DLP programs, and that the potential improvement from microarchitecture mechanisms targeted at these overheads is quite large. These large potential speedups are not a result of poor starting baseline. As mentioned in Chapter 6, for many applications the TRIPS processor is up to 2X better than a 4-issue aggressive out-of-order superscalar processor like the Alpha 21264.

8.3.3 Summary

The quantitative analysis and the detailed program characterization show that DLP programs share a set of common attributes. The quantitative analysis shows that building microarchitecture mechanisms targeted at these specific attributes can provide significant improvements. For example, if we reduced all of the fetch overheads for *FFT*, a 4X improvement in performance is possible. A 9X improvement in performance is possible for *LU* if all the overheads in memory accesses are removed. The percentage of critical cycles devoted to a type of microarchitecture event directly conveys the speedup possible by removing the overheads associated with that event. For example, 88.9% of the cycles in *LU* are spent in memory accesses, which implies a maximum speed of $(100 - 88.9\%)/100 = 9$. Secondly, since this is an analysis based on the critical path of microarchitecture events, it is likely that the performance improvement from multiple mechanisms will be additive. Finally, by subtly changing the programming and execution model, it is possible to achieve speedups beyond what is possible by simply reducing microarchitecture overheads. For example, some programs with fine-grained concurrency can be dramatically speeded up using decoupled execution between "threads" that the MIMD paradigm provides.

Examining the workloads and the distribution of DLP attributes among these workloads, we observe that our benchmark suite captures an important and large subset of the DLP space. However, it is not clear that the applications we have individually isolate each attribute in the DLP space. For example, although *FFT* shows a significant instruction-fetch bottleneck, it is not clear there is a fundamental behavior of that program that makes it instruction-fetch limited. One area of future work is to determine a mapping of programs to specific single microarchitecture events and identify specific program structure and code patterns that create microarchitecture bottlenecks.

This analysis of the microarchitecture critical path was based on the TRIPS microarchitecture. However, we grouped microarchitecture events specific to the TRIPS design like *register read instruction delay* into high-level processor events such as fetch, register access, and memory access. Our analysis of these high-level processor events showed fundamental bottlenecks that hinder the performance of DLP workloads. This analysis is targeted at such high-level processor events to abstract out the specifics of the TRIPS microarchitecture and hence the conclusions of this study can be broadly applied to other conventional processors. While the quantitative improvements may differ, we expect to see similar trends and qualitative results.

8.4 Data-Parallel Microarchitectural Mechanisms

The program analysis presented in Section 8.2 provided us with insight into program behavior and the critical path analysis in the previous section quantified the bottlenecks in the execution core, instruction control, and memory system. In this section we describe the microarchitecture mechanisms we developed based on these insights. Figure 8.2 shows a block diagram of an abstract microarchitecture. We explain the polymorphous mechanisms in terms of these abstract resources and specifically in the context of the TRIPS processor. The mechanisms proposed in this study are not implemented in the TRIPS prototype chip.

8.4.1 Memory System Mechanisms

The memory system in a data-parallel architecture must support high bandwidth regular memory access and low latency irregular memory accesses. Our microarchitecture bottleneck analysis showed that memory accesses on average account for 30% of the critical path and optimized mechanisms could potentially pro-

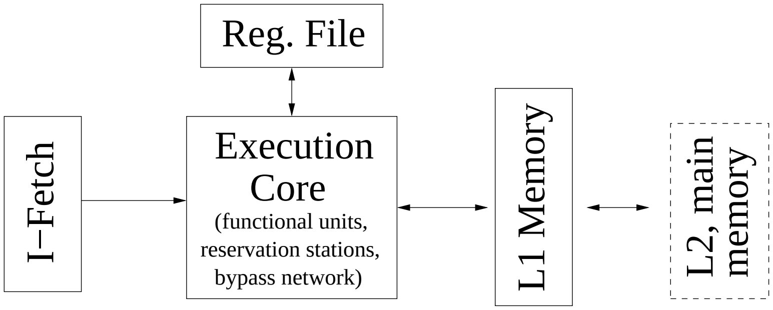


Figure 8.2: Microarchitecture block diagram.

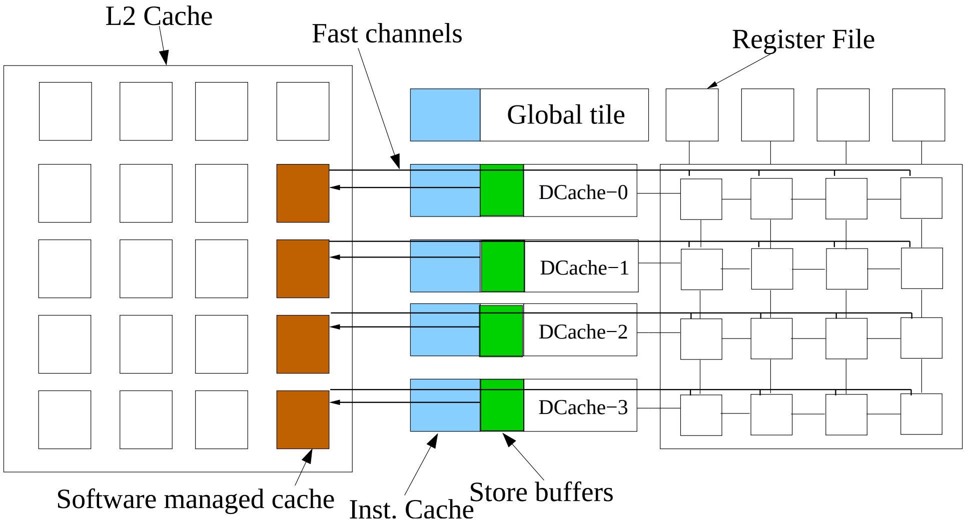


Figure 8.3: Memory system mechanisms. Software managed cache, fast channels and store buffers.

duce speedups up to 9X for the DLP programs. We propose a software managed cache and a hardware managed cached memory system for these accesses respectively.

Software managed cache: Figure 8.3 shows the configuration of the memory system that provides a high-bandwidth access for regular access patterns. Portions of the secondary-level cache banks can be reconfigured as a fully software managed cache (SMC). In this configuration, the hardware replacement scheme and tag checks in these cache banks are disabled. The SMC banks each contain a DMA engine that is explicitly programmed by software. These banks are exposed to and are fully managed by the programmer or compiler. Only the regular memory accesses (statically identifiable by the compiler) use the SMC, and they also bypass the L1-cache since temporal locality is poor. Using the data tiles which form the L1-cache is also possible because managing coherency at that level becomes a challenge. The programming abstraction and interface used in Imagine's Stream Register File (SRF) [86] may be used to manage this SMC. Providing such software managed caches (referred to as a stream register file or SRF) is a natural configuration to exploit the regular access patterns while providing high bandwidth. The DMA engines are used to essentially prefetch large blocks of memory into these banks and provide high bandwidth transfer from main memory into the SRF.

Wide loads: Overhead and latency to access the SMC can be reduced by using a `LMW` (load multiple word) instruction for reads. An `LMW` instruction issued by one ALU fetches multiple contiguous values and sends them to many ALUs or multiple reservation stations in the same ALU in a single row inside the array. To reduce

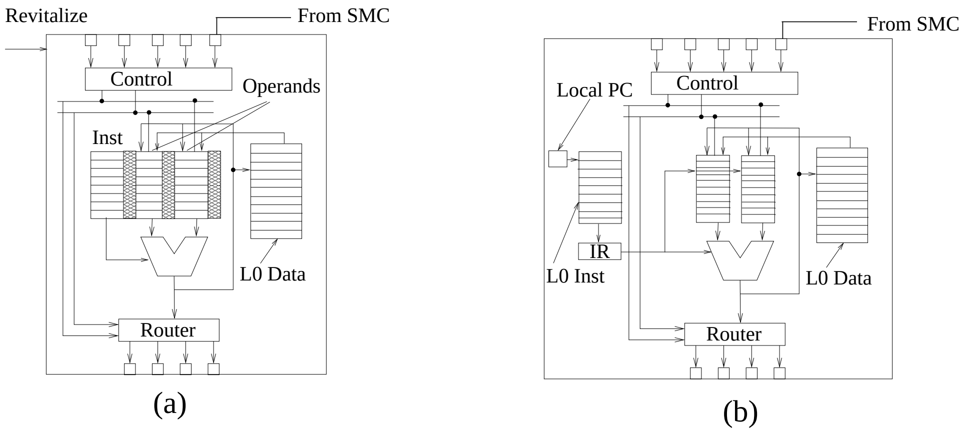


Figure 8.4: Execution core and control mechanisms. a) Instruction, operand revitalization and L0-data storage. b) Local PC and L0-instruction store to provide MIMD execution.

the write port pressure, a store buffer coalesces stores from different nodes together before writing them back to the SMC.

High-bandwidth streaming channels: To deliver these operands at a fast rate to the execution core, dedicated channels are provided from the SMC banks to a corresponding row of ALUs. The array based design provides a natural partitioning of the cache banks to rows of ALUs.

Cached L1-memory: Irregular memory accesses can be efficiently handled by using the level-1 cache and those banks in the level-2 not configured as SMC banks. In applications such as graphics rendering, such a caching mechanism for the irregular texture lookups can provide low latency access [65].

8.4.2 Instruction Fetch and Control Mechanisms

The branching behavior of data-parallel kernels dictate instruction fetch and control requirements which are: (1) repeated fetching and mapping of kernel instructions to reservation stations, resulting in instruction cache pressure and dynamic cache access power, and (2) MIMD processing support for kernels that exhibit fine-grained data dependent branching. To avoid repeatedly fetching instructions of a loop, the ALUs are enhanced to reuse instructions for successive iterations reading from a local storage. To efficiently support data dependent branching, each ALU is augmented with a local program counter (PC).

Instruction revitalization: In the TRIPS processor, the ALUs already contain local instruction storage. To efficiently support the execution of loops, we augment the ALUs with support for re-using instruction mappings for successive iterations of a loop. This mechanism, which we call *instruction revitalization*, works as follows: before the start of a kernel, a *setup block* executes a *repeat* instruction specifying the run-time loop bounds of the kernel which is saved to a special hardware count register `CTR`. Then the instructions of the kernel are mapped to the execution core and execute their first iteration. When the iteration completes (determined by the block control logic), the `CTR` register is decremented. If the counter has not yet reached zero, the block control logic broadcasts a global revitalize signal to all the nodes in the execution array - which resets the status bits of the instructions in the reservation stations, priming them for executing another iteration. When the `CTR` register reaches zero, the next kernel's execution commences.

To amortize the cost of the global revitalize broadcast delay, blocks are unrolled as much as possible, as determined by the number of the reservations stations, so as to reduce the number of revitalizations. Figure 8.4a shows the datapath and control path modifications added by this mechanism. The shaded regions next to the reservation stations indicate the status bits required for revitalization. In the TRIPS processor, using instruction revitalization provides a vector/SIMD-like architecture model.

Local program counters: To support fine-grained data dependent branching, the execution core is configured as a MIMD processing array by adding local PCs at the ALUs. To simplify the datapath we also add a separate *L0 instruction storage* from which instructions are fetched and executed sequentially. (A slightly more complex, but area efficient implementation is to re-use the local instruction storage already present in the ALUs and use the PC to read this storage.) Prior to executing kernels in a MIMD mode, their instructions are loaded into this store by executing a *setup* block, which copies instructions from memory into this storage and resets the local PC to zero at every ALU. Once this *setup* block terminates, the array of ALUs begin executing in MIMD fashion. Each node independently sequences itself by fetching from its local instruction store. The operand storage buffers are used as read/write registers, providing a simple in-order fetch/register-read/execute pipeline. Figure 8.4b shows a schematic of the modified ALU datapath to support such a MIMD model. While this MIMD model has a one time startup delay, instruction revitalization incurs a revitalization delay between every iteration.

Multiple nodes can be aggregated together to execute one iteration of a kernel in this MIMD model, providing a logical wide-issue machine for each iteration of the kernel, using the inter-ALU network for fine-grained ALU-ALU synchronization. In this configuration the ALU array can thus be partitioned into multiple dynamically issued cores. Another mode of operation is to execute different kernels on the ALUs, passing values using between them through the inter-ALU network. In real-time graphics processing for example, a rendering pipeline can be implemented by partitioning the ALUs among vertex processing, rasterization, and fragment processing kernels. Since the ALUs are homogeneous and fully programmable, the partitioning of ALUs can be dynamically determined based on scene attributes. This strategy overcomes one of the limitations of current graphics pipelines in which the vertex, rasterization and fragment engines are specialized distinct units.

8.4.3 Execution Core Mechanisms

Efficient scalar operand and indexed scalar operand access must be supported for data-parallel execution. For large, statically unrolled loops, reading values from the registers for each iteration of the loop is expensive in terms of power, register file bandwidth, and other overheads of register file access. Using the memory system for indexed scalar operands incurs cache access overheads and consumes cache bandwidth. Two mechanisms implemented at the execution core support these two types of accesses efficiently.

Operand revitalization: This mechanism reuses register values once they have been received at an ALU, providing persistent register-file like storage at each reservation station. Successive iterations of the loop reuse the values from the reservation stations instead of accessing the global register file. To implement operand revital-

Attributes	Mechanisms	Implemented at	Benchmarks that benefit
Regular memory access	Software managed streamed memory	L2 Memory	All
Irregular memory access	Cached memory subsystem	L1 Memory	fragment-simple, fragment-reflection
Scalar named constants	Local operand storage (Operand revitalization)	Execution core, Register file	convert, dct, highpassfilter, md5, rijndael, all graphics programs
Indexed named constants	Software managed L0 data store at ALUs	Execution core	blowfish, rijndael, vertex-skinning
Tight loops	Local instruction storage (Instruction revitalization)	Execution core, Instruction fetch	All
Data dependent branching	Local program counter control	Instruction fetch, Execution core	vertex-skinning, anisotropic-filtering

Table 8.5: Data-parallel program attributes and the set of universal microarchitectural mechanisms. Mechanisms in parenthesis indicate TRIPS specific implementations.

ization we add status bits to the reservations stations, as shown in Figure 8.4a.

L0 data storage: A software managed L0 data storage at each ALU provides support for indexed scalar constants (one example is the lookup tables used in encryption kernels). Figures 8.4a and 8.4b show the L0 data store, which is accessed using an index computed by some instruction with the result being written to the reservation stations. The index to read the L0 data store is provided by the ALUs and the results are written back into the local registers as shown. For the applications we examined, 2KB was sufficient to store all such constants.

8.4.4 Summary

Table 8.5 summarizes the program attributes that we identified in our program characterization study and maps these to the mechanisms we described above.

The first column of Table 8.5 lists these attributes. The second column lists the proposed mechanisms targeted at different microarchitecture components as shown in the third column. The last column lists the benchmarks that benefit from each mechanism. Two mechanisms are implemented in the memory system: (1) a software managed streamed memory subsystem is used to support high bandwidth regular memory accesses, and (2) a hardware managed cached memory subsystem is used to support efficient irregular memory accesses. The execution core is enhanced with additional local operand storage to efficiently support named scalar operand accesses, and an additional software managed local data storage for accessing indexed named constants. Finally examining control behavior, instruction storage at each ALU in the execution core is added for supporting short simple loops, and a local program counter at each ALU is added to provide data dependent branching behavior.

While we described these mechanisms using the TRIPS processor as the baseline, they are universal and applicable to other architectures. The SMC, store buffer and the `LMW` instructions can be added in a straightforward manner to conventional wide-issue centralized or clustered superscalar architectures by adding direct channels from the L2-caches to the functional units and augmenting the pipeline to wakeup instructions dependent on the loads when their operands arrive from the SMC. The Tarantula architecture provides similar such support for transfers from the L2 memory to the vector register file, using hardware techniques to generate conflict free addresses to different banks in memory, in contrast to our approach of packing all the regular accesses in a single bank. To support indexed scalar access and irregular memory accesses in this architecture, the L1-cache memory must be addressable using special scatter/gather instructions. Most conventional superscalar processors provide good support for L1-cache memories.

The reservation stations in TRIPS have a one-to-one correspondence to reservation stations in superscalar architectures and both the instruction and operand revitalization mechanisms can be applied to provide instruction and operand re-use. To achieve instruction sequencing efficiency, many DSP processors have implemented zero-overhead branches in different ways to support tight loops [50].

To provide MIMD support, local PCs are added and the local ALU control logic modified to fetch from a local instruction store buffer. Conventional SIMD and vector cores conversely have no local storage and thus must be augmented with a local PC and storage buffers to provide a MIMD model of execution. While adding such local storage goes against the spirit of polymorphism and could dramatically increase the design complexity of vector and SIMD machines, these modifications increase the domain space they can target.

8.5 Results

This section presents the compilation strategy, simulation methodology, and the performance evaluation of the mechanisms. The results focus on evaluating and measuring the following: (1) performance improvement provided by each mechanism, (2) benefit from different mechanisms for each application, (3) performance of a flexible architecture constructed using a combination of the mechanisms, and (4) this flexible architecture's performance relative to specialized architectures.

8.5.1 Simulation Methodology

For the ILP and TLP evaluation study we used the *tsim-proc* cycle accurate simulator. For the evaluation of the DLP mechanisms we use a different infrastructure, primarily because modifying *tsim-proc* to model all the mechanisms would make it too slow. Furthermore, the simulator itself is too closely tied to the TRIPS prototype implementation and is not easily extensible. We use a more abstract simulator, which has been described by Desikan [41] as the GPA simulator, that models the TRIPS processor. This simulator uses binaries generated by the IMPACT compiler and translates instruction into a TRIPS-like instruction set, and uses a scheduler that has similar heuristics to the TRIPS scheduler. The different mechanisms were integrated into this simulator for the performance experiments. Appendix A describes more details on this simulation infrastructure and compares this simulator to *tsim-proc*.

All the programs were hand-coded in a TRIPS like instruction set to exploit these data-parallel mechanisms and then simulated. Since we did not have sufficient infrastructure and datasets for a realistic simulation of *anisotropic-filtering*, we exclude it from all our performance tables and figures. All the opcodes used are opcodes present in the TRIPS ISA, used in the prototype chip. The only difference between this TRIPS-like ISA and the TRIPS ISA is that the file formats for the binaries. Hence some instruction cache behavior would be different. Where possible we statically unrolled the kernels to fill up the instruction storage across the ALUs. We measure relative speedups in terms of execution cycles between the baseline and the different machine configurations. The simulations assumed that

Benchmark	**Ops/cycle**	**Benchmark**	**Ops/cycle**
convert	3.5	fragment-reflection	1.0
dct	2.6	fragment-simple	0.7
highpassfilter	1.9	vertex-reflection	1.3
fft	0.9	vertex-simple	1.3
lu	0.2	vertex-skinning	1.4
md5	0.8		
blowfish	1.2		
rijndael	1.9		

Table 8.6: Performance on baseline TRIPS.

all data were resident in the software managed cache (SMC) or L2 storage for all applications. Except for *LU*, the datasets of all applications fit entirely in the SMC.

8.5.2 Baseline TRIPS Performance

Our baseline configuration models the TRIPS prototype chip with the GPA simulator. We assume each data cache bank is connected to a 64KB SMC bank. The functional unit and cache access latencies are configured to match an Alpha 21264. Each node in the processor consists of an integer ALU, integer multiplier, and an FPU with add, multiply, and divide capability.

Table 8.6 shows the performance of the baseline measured in terms of number of useful computation operations sustained per cycle, not including overhead instructions like address compute and load and store instructions. Only the DSP programs sustain a very high computation throughput, averaging about 3 ops/cycle, while all other applications sustain low throughputs, averaging about 1 op/cycle.

Since the baseline TRIPS processor is optimized for ILP, converting the data-level parallelism in these applications to ILP results in inefficiencies for DLP

Config.	**L0 store**		**Revitalization**		**Architecture model**
	Inst.	**Data**	**Inst.**	**Ops.**	
S	N	N	Y	N	SIMD
S-O	N	N	Y	Y	SIMD+ scalar constant access
S-O-D	N	Y	Y	Y	SIMD+ scalar constant access+ lookup table
M	Y	N	N	N	MIMD
M-D	Y	Y	N	N	MIMD+lookup table

Table 8.7: Machine configurations.

programs. For example, loops cannot be sufficiently unrolled to provide large enough blocks to efficiently utilize the array of ALUs, and every scalar operand or memory reference must proceed through shared structures such as the L1 cache and the common register file. Since many DLP programs have large demands on these resources, the limited bandwidth prevents the architecture from achieving its potential performance.

8.5.3 Configuration of Mechanisms

The mechanisms described in Section 8.4 can be combined in different ways according to application requirements to produce as many as 20 different run-time machine configurations of a single flexible architecture. The frequency of each type of memory access, the control behavior of the kernels and the instruction size of kernels, measured in Table 8.2 and 8.3 determine the ideal combination of mechanisms on the TRIPS processor. In this dissertation we focus on five machine configurations, shown in Table 8.7, that cover the application set we examined.

In all five configurations, one memory bank per row is configured to be used as a software managed cache. The SMC banks use the store buffers and the high speed channels to communicate with the execution core. We describe the five configurations in detail below:

- **SIMD machine:** Combining software managed memory system with an instruction revitalization mechanism creates a baseline model that is similar to SIMD and vector machines. Instruction revitalization adds the support for instruction and control efficiency that make SIMD and vector machines efficient at DLP. The reservation stations distributed accorss the tiles can be thought of as forming a distributed vector register file and the instructions mapped across the different tiles form one large vector instruction.

- **SIMD + scalar operand access:** This baseline machine (**S**) can be augmented with operand revitalization to create the **S-O** machine. This configuration optimizes the injection of values into the execution array.

- **SIMD + scalar operand + lookup table access:** The **S-O-D** machine adds local L0 data storage to each ALU of the S-O machine. This configuration departs the most from the spirit of polymorphism as it adds additional storage elements, beyond simply modifying control logic.

- **MIMD:** Combining the memory system with local PCs creates a baseline MIMD machine (**M**). In addition the control logic at the ALUs is augmented to sequence instructions instead of execution in pure dataflow fashion.

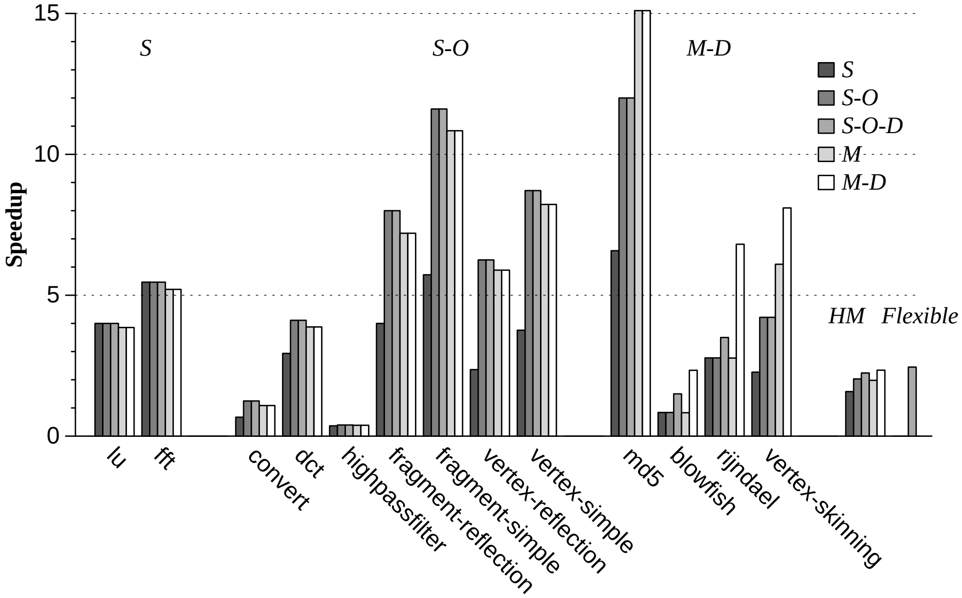


Figure 8.5: Speedup using different mechanisms, relative to baseline architecture. Programs grouped by best machine configuration.

- **MIMD + lookup table access:** Addition of local L0 data storage creates to previous configuration creates the **M-D** machine.

8.5.4 Performance Evaluation

Figure 8.5 shows the application speedups obtained by these different machine configurations relative to the baseline. The following paragraphs classify the applications by their preferred configurations. Two benchmarks preferred the S, seven preferred the S-O and four preferred M-D configuration.

- **SIMD execution (S):** *Fft* and *LU* are vector-oriented benchmarks and require high memory bandwidth and high instruction fetch rate. Compared to the baseline a four-fold speedup is achieved because of the higher ALU utilization and higher memory bandwidth of the **S** configuration. Adding other mechanisms does not improve performance further, and the routing overhead of MIMD execution degrades performance slightly.

- **SIMD + scalar operand access (S-O):** The performance of many applications is dictated by the frequency of scalar operand access (35 constants in *vertex-reflection* for example). These perform best on the **S-O** machine configuration as shown by the set of 7 programs in Figure 8.5.

- **SIMD + scalar operand + lookup table access (S-O-D):** *Blowfish, and rijndael* which use reasonably large lookup tables show speedups of 27% and 80%, respectively, over the **S-O** configuration, but perform worse than the **M-D** machine.

- **MIMD (M):** The baseline MIMD configuration degrades performance somewhat relative to **S-O-D** for all applications except *vertex-skinning*. This degradation arises because in the MIMD model the load instructions from each ALU must be routed through the network to reach the memory interface. In the previous three **SIMD** configurations, synchronized at block boundaries, a multi-word load instruction could be placed near the memory interface, to behave like a vector fetch unit. Since each node operates independently in the MIMD model, such a schedule is not possible.

- **MIMD + lookup table access (M-D):** The MIMD machine with lookup table support performs best for *md5*, *blowfish*, *rijndael*, and *vertex-skinning*. With local looping control, these programs require far less instruction storage and hence can be unrolled more aggressively providing more parallelism. Because *vertex skinning* uses data dependent branching, the overheads of predicated execution (or conditional vectors) are also removed.

- **Flexibility:** The last single bar labeled *Flexible* in Figure 8.5 shows the harmonic mean of speedups achieved by a flexible architecture when a subset of mechanisms are combined according to application needs (running *fft* and *LU* on **S**, *convert* through *vertex simple light* on **S-O**, and the rest on **M-D**). Averaged across the different applications, this flexible dynamic tuning provides 55% better performance over a fixed **S** configuration, 20% better than fixed **S-O** and 5% better than a fixed **M-D** machine.

8.5.5 Comparison Against Specialized Architectures

Table 8.8 shows the results of a rough comparison between the performance of the configurable TRIPS architecture to published performance results on specialized hardware. Columns 2 and 3 show performance, column 4 shows the performance metrics (which vary), and column 4 describes the specialized hardware. For each of the applications we picked the best combination of the mechanisms on the TRIPS baseline. When appropriate, we normalized the clock rate of TRIPS to that of the specialized hardware. Scaling the clock does not violate any microarchitecture assumptions, since the TRIPS processor is designed for clock rates at least

Benchmark	**Performance**		**Units**	**Reference h/w**
	TRIPS (clock normalized)	**Specialized h/w**		
DSP kernels				
convert	4754	960	iterations/sec	MPC 7447, 1.3Ghz (embedded processor)
highpassfilter	705	907	iterations/sec	
dct	8.5	8.2	ops/cycle	Imagine [135] (multimedia processor)
Scientific computing kernels				
fft	14.4	28	ops/cycle	Tarantula [48] (vector core)
lu	10.6	15	ops/cycle	
Network processing kernels				
md5	14.6	-	cycles/block	Cryptomaniac [172]
blowfish	6	80	cycles/block	
rijndael	12	100	cycles/block	
Graphics processing kernels				
			(millions)	
fragment-reflection	86	-	fragments/sec	Nvidia QuadroFX 450Mhz (graphics processor)
fragment-simple	193	1500	fragments/sec	
vertex-reflection	434	-	triangles/sec	
vertex-simple	418	64	triangles/sec	
vertex-skinning	207	-	triangles/sec	

Table 8.8: Performance comparison of TRIPS with DLP mechanisms to specialized hardware.

as high as conventional designs and very likely higher than the typical high FO4 designs of these specialized processors.

On the signal processing codes, the TRIPS core in the *S-O* configuration, is up to 5 times faster than the MPC 7447, an embedded processor, with the improvement coming from the 4X higher issue-width (4 vs. 16). The TRIPS core contains roughly half the number of functional units as the Imagine architecture and performs roughly a factor of two worse on *dct*.

For the scientific codes we compare performance to the Tarantula architecture. The TRIPS *S* configuration is store bandwidth limited and about a factor of two worse than the Tarantula architecture. The TRIPS peak memory bandwidth from the processor to the memory system for stores is 4 words/cycle for an execution array with 16 execution units, whereas Tarantula allows 32 words/cycle on an execution array with 32 execution units.

For the network processing programs we compare performance to Cryptomaniac, a programmable specialized network processor. By exploiting the extensive data-level parallelism in network flows, the TRIPS *S-O* and *S-O-D* configurations perform an order of magnitude better than specialized hardware, where the packets are processed serially (smaller numbers in the table for these programs indicates better performance). Cryptomaniac could also potentially exploit concurrency across packet flows, and in fact many network processors do exactly that by providing multiple simple cores on chip and assing each core a network stream.

We programmed the graphics kernels for the NVIDIA QuadroFX chip and measured performance on a 2.4 GHz Pentium4 based system. In the *vertex-simple* graphics application, TRIPS outperforms the dedicated hardware primarily because of the much higher issue width and functional unit count. On *fragment-simple* on the other hand the specialized hardware outperforms TRIPS by roughly 8X. Although the exact details on the number of functional units (fixed point + floating point units) on the QuadroFX are not publicly disclosed, we believe part of this high performance can be attributed to the larger number of functional units. The other graphics processing kernels are more complex (using more instructions, more constants, and data dependent branching in one case) than the two we benchmarked, and will perform at best as well as the other kernels, and likely poorer.

8.6 Summary

In this chapter we presented a comprehensive treatment of programs covering a large spectrum of the DLP application space, including signal processing, scientific, network/security, and real-time graphics applications. While there may be DLP applications outside these domains, the four studied in this dissertation provide comprehensive coverage over the application space. We identified the key memory, control, and computation demands of DLP applications and characterized the behavior of the DLP application suite.

We then proposed a set of complementary universal microarchitectural mechanisms targeted at the memory system, instruction control, and execution core, that can support each type of DLP behavior. For the memory system, we proposed a streamed software managed cache memory along with a hardware managed level-1 cache. For the execution core and instruction control we proposed local operand storage, local instruction storage, a software managed local storage, and local program counters at each ALU site. These mechanisms can be combined in different ways based on application demand and are powerful enough to provide both a SIMD and MIMD execution model on the same substrate. We found the approach of customizing the architecture resulted in 5%–55% better performance than a fixed yet scalable architecture. The approach in this dissertation of customizing the architecture to the application has similarities to the philosophy of Custom-fit processors [54], but the customization we propose enables different execution models on the same substrate and can be performed after fabrication. When compared to application-specific processors in each of the domains, the architecture built using the mechanisms in this dissertation achieves performance in a similar range, when normalizing for clock rate and ALU count. While each application specific processor performs well in its own domain, none have significant flexibility to perform well on DLP applications outside its domain.

The mechanisms that we propose are not strictly limited to the TRIPS processor described in this dissertation. For example, the hybrid of SIMD and fine-grained MIMD execution models is a reasonable goal for other DLP architectures. Future systems that must execute multiple classes of DLP applications will benefit by implementing all of the mechanisms and dynamically configuring the architecture based on application needs. However, when only a subset of DLP behavior needs to be supported, the flexibility can be sacrificed for simplicity by implementing a subset of the mechanisms on a fixed architecture by matching the mechanisms to the application attributes.

Chapter 9

Conclusions

Processor architects today are faced by two daunting challenges: emerging applications with heterogeneous computation needs and technology limitations of power, wire-delay, and process variation. Designing multiple application specific processors or specialized architectures introduces design complexity, a software programmability problem, and reduces economies of scale. In this dissertation, we introduce *architectural polymorphism* to build scalable processors that provide support for heterogeneous computation by supporting different granularities of parallelism on a single processing substrate. The basic idea in polymorphism is to configure coarse-grained microarchitecture blocks to provide an adaptive and flexible processor substrate. Technology scalability is achieved with scalable and modular microarchitecture blocks.

9.1 Summary

In this dissertation, we identified the granularity of parallelism as the fundamental difference between application classes and use it categorize application heterogeneity with respect to processor architecture. The three granularities of parallelism are instruction-level, thread-level, and data-level parallelism. To provide architectural support across all these types of parallelism, we propose architectural polymorphism driven by three main principles: adaptivity across these granularities of parallelism, economy of mechanisms, and microarchitectural reconfiguration at a coarse granularity.

We use the dataflow graph as the unifying abstraction layer across these three types of parallelism. We introduce EDGE ISAs, a class of ISAs, as an architectural solution for efficiently expressing parallelism for building technology scalable architectures. All programs are expressed in terms of dataflow graphs and directly mapped to the hardware which is partitioned depending on the granularity of parallelism.

EDGE ISAs: EDGE ISAs encode dependences directly in the program binary and employ a block atomic execution model. The explicit dependence encoding efficiently expresses the dataflow graph (and hence concurrency), obviating the need for complex hardware to rediscover parallelism. The block atomic execution model, raises the granularity of execution and state management in the hardware and eliminates instruction-level overheads. Instead of tracking architectural change at an instruction level which leads to a lot of instruction-level overheads, architectural change occurs at a block-level, reducing the frequency of branch predictions, register reads and writes, and register renaming.

TRIPS: We developed the TRIPS architecture as an implementation of EDGE with a heavily partitioned and distributed microarchitecture implementation to achieve technology scalability. The two most significant features of the TRIPS microarchitecture are its heavily partitioned and modular design, and the use of microarchitecture networks for communication across modules.

Polymorphism: This dissertation introduces architectural polymorphism: the capability to configure the hardware at run-time to perform different functions. Unlike reconfigurable architecture that synthesize complex logic from primitive functions, the polymorphism principle is to build coarse-grained reconfigurable microarchitectural blocks whose function can be changed at run-time. We used the TRIPS architecture as the baseline for developing and implementing these polymorphous mechanisms. The TRIPS architecture is a modular design with well defined microarchitecture blocks and is a technology scalable design, thereby serving as a good baseline starting point for implementing polymorphism. We proposed and evaluated mechanisms targeted at three processor resources: the execution core, control flow unit, and memory system.

Results: Our performance results show that the TRIPS microarchitecture can sustain good instruction-level concurrency, despite the potential overheads of its distributed protocols. On a set of hand-optimized kernels, the processor sustains IPCs in the range of 4 to 6, and on a set of highly data parallel benchmarks with compiler generated code IPCs are in the range of 1 to 4. On the EEMBC and SPEC CPU2000 benchmarks, with compiler generated code we see IPCs in the range of 0.5 to 2.3, with an average IPC of 1.1 for the EEMBC suite and 1.6 for the SPEC

CPU2000 suite. On hand optimized microbenchmarks, the TRIPS processor is up to 4 times better than an Alpha 21264. With compiler generated code for large sophisticated benchmarks like the EEMBC and SPEC CPU2000 benchmarks, the TRIPS processor performs worse than the Alpha 21264 in most cases.

Hand-optimized versions of the EEMBC benchmarks perform up to 8 times better than the Alpha 21264 and many benchmarks share several of the same optimizations. Some of these hand optimizations, which include better instruction merging, load/store dependence elimination through better register allocation, and scalar instruction-level optimizations (reducing arithmetic computation tree heights) are not unreasonable to implement in the compiler. These are currently hand optimization and not yet in the compiler for two reasons: 1) the heuristics applied for these optimizations vary from benchmark to benchmark and are at times based on examining microarchitecture critical path events, and 2) our cycle accurate simulators are too slow and we expect to understand more of the hardware's behavior on complex codebases when we have manufactured chips in the lab. As the compiler matures and we develop a better understanding of the heuristics, we expect more of these optimization to be integrated into our compiler and the compiler generated code performance to improve.

The polymorphous mechanisms proposed in this dissertation are effective at exploiting thread-level parallelism and data-level parallelism. When executing 4 threads on a single processor, significantly higher levels of processor utilization are seen, IPCs are in the range of 0.7 to 3.9 for an application mix consisting of EEMBC and SPEC CPU2000 workloads. Compared to an average IPC of 1.1 and 1.6, these application mixes have much higher IPCs–2.2 when running with 2 applications concurrently, and 3.1 when running with 4 applications.

When executing programs with data-level parallelism, compared to an execution model of extracting only ILP in the TRIPS processor, the DLP mechanisms provide average speedups of 5.6 across a set of DLP workloads. The speedup provided by the individual mechanisms range from 1 to 15.2. The polymorphous mechanisms enable the TRIPS architecture to match the performance of specialized processors targeted at different types of DLP workloads. Specifically, the polymorphous mechanisms allow the configurable TRIPS chip to match the performance of best-of-breed DSP chips, graphics chips, and vector chips on workloads specialized for each.

9.2 Discussion

We have developed a prototype chip that implements the TRIPS ISA and at the time of this dissertation, we expect systems back at the end of Fall 2006. In 2001 we started with promising results based on high-level simulation. The implementation of the prototype shows that those ideas are feasible, and the microarchitecture networks show that a block atomic model can be effectively implemented by a physically distributed design.

These distributed protocols have enabled us to construct a 16-wide, 1024-instruction window, out-of-order processor, which works quite well on a small set of regular, hand-optimized kernels. We have not yet demonstrated that code can be compiled efficiently for this architecture, or that the processor will be competitive even with high-quality code on real applications. Once systems are up and running in the Fall of 2006, a detailed evaluation of the capabilities of the TRIPS design will help understand the strengths and weaknesses of the system and the technology and answer these questions.

In this dissertation, we have made a strong case for polymorphism based on a homogeneous computing substrate to satisfy the computation needs of future applications that are likely to have heterogeneous computation needs. We believe this approach is superior to building a heterogeneous system composed of multiple specialized processors. For designers who wish to build polymorphous systems, the three main challenges are VLSI design complexity, software complexity, and technology constraints of performance, power, area, and reliability–all of which translate into market constraints.

9.2.1 VLSI Design Complexity

In terms of VLSI design complexity, the homogeneous approach has definite advantages. In this dissertation, we introduced a principled approach of using polymorphism to achieve design convergence and have focused on providing diverse functionality using an economy of mechanisms, driven by a detailed understanding of program behavior and quantitative analysis. For example, we demonstrated a clear instruction control bottleneck on scientific computing kernels like *fft* and *LU* decomposition by program analysis. Our critical path analysis showed that more than half the program cycles are spent in feeding the processor core with instructions. This motivated control enhancements that enabled fetched instructions to be reused in the processor core without introducing any new storage structures. Overall, the number of mechanisms to cover ILP, TLP, and DLP are few in number, well defined, and targeted at specific resources in the processor. Implementing these would be simpler than building multiple cores on chip, each core tailored for a type of application.

As an illustrative case study, we compare the Tarantula processor, which is a heterogeneous design, to TRIPS. The Tarantula processor comprises a 32 wide vector core and a high performance out-of-order EV8 core integrated on a single chip [48], whereas the polymorphous TRIPS design includes two homogeneous polymorphous processor cores. The specific benefits of polymorphism in the TRIPS are in design reuse in the processor core, the memory system, and the register files.

- In the TRIPS approach there is significant savings and reuse in datapath design since one core is replicated instead of having to design two different cores.

- The Tarantula architecture provides a pure vector model at significant design cost. Tarantula provides global synchronization between the different vector lanes with partitioned vector registers and optimized accesses to the regular L2 cache for vector loads. The designers went to great lengths to provide the high bandwidth required out of the L2 cache. In TRIPS, we simplified the memory system and instead provide support to create a software managed memory system by reconfiguring the L2 cache banks as scratchpad memories. While the Tarantula approach to allow vector access to the L2 cache includes a complex conflict free address generation scheme to maximize bandwidth [145], to create scratchpad memories at each TRIPS memory tile, the tags checks are simply disabled. The Cell processor uses a similar approach to manage memory.

- Unlike Tarantula which contains vector register files that need to be read and written for every instruction, we showed (but did not implement in the prototype chip) polymorphous mechanisms that can use the reservation stations closely integrated with each ALU to create vector register file like behavior with superior bypassing capability.
- Since Tarantula is a vector processing core, accesses to the L1 caches are disabled, consequently programs that require lookup tables, large number of constants and other irregular data structures perform poorly. In the TRIPS approach, an application can chose to continue using the L1-caches for such irregular accesses, while using the software managed memory for high-bandwidth regular memory accesses.

This dissertation did not address the verification complexity of these mechanisms or show how to limit the interaction between these mechanisms and thus achieve verification closure. The mechanisms are by definition unrelated and can be used separately or together. For example, the five DLP mechanisms result in about 20 processor configuration which presents a reasonably daunting verification challenge. With a heterogeneous solution, the number of specialized designs is known and the verification methodology for them is well defined. The verification complexity of such a heterogeneous design compared to a polymorphous design is an interesting question to address while deciding on which solution to pick. While this dissertation leaves the question open, we do not view it as an intractable or hard challenge. The TRIPS prototype chip implements a limited amount of such polymorphous support where the mechanisms can be dynamically chosen, for example, the "multithreaded mode" of the processor, a single-block execution mode of the processor, and the configuration of the memory tiles as scratchpad memories. We verified these mechanisms and *modes* of the processor through randomized testing by generating random programs and deciding on the processor *modes* through randomization. The level of coverage achieved in this process leads us to believe that the verification is not much more difficult than verifying multiple heterogeneous cores.

9.2.2 Software Complexity

Designing, developing, and compiling applications with heterogeneous computation needs presents challenges for the entire software stack. When the target is a heterogeneous processor with multiple specialized processors, one must decide which application is best suited for which processor. When the target is a homogeneous processor with polymorphous capabilities, one must decide on the configuration of the different microarchitectural blocks. Is compiling for such homogeneous systems more complex than compiling for heterogeneous systems?

Some software design issues are common to both systems, namely, determining application behavior, determining the granularity of the parallelism, and mapping of processor capability to the application. On the other hand, some software decisions are different because the two systems are so radically different. Examples include the following: 1) while compiling and designing for heterogeneous systems knowing the application mix is important, 2) migrating applications from one specialized core to another can pose a challenge since each core is tuned to a specific type of application, and 3) application phase behavior, in which the type of parallelism in a single application changes during its run time, can be hard to manage. On the other hand, designing for homogeneous systems poses different challenges: 1) determining the mapping of the mechanisms to application behavior, and 2) expressing and exposing the polymorphous microarchitecture features to the compiler.

In this dissertation, we did not address this software complexity challenge. We only showed that among a set of possible configurations, there was a natural and preferred configuration for some applications. We did not address how the compiler or run-time system can determine these properties or the ideal configuration.

These software design questions must be addressed irrespective of whether designers choose to building heterogeneous systems or homogeneous systems. Recent research in compilers and programming languages points to promising directions that may address this software complexity challenge. Over the years, several application specific compilers have been proposed to deal with growing processor complexity. Application specific compilation that is aware of program properties can outperform general purpose compilation. FFTW is perhaps the best know example of application specific compilation [57]. Other recent examples include

FLAME [62] and ATLAS [170] targeted at linear algebra, SPIRAL [128] which uses a dynamic programming approach to optimize the compilation of DSP routines, and the Broadway compiler meant for domain specfic libraries and specifically scientific computing libraries [64]. Programming language efforts include Streamit [60] targeted at streaming and multimedia programs, Cg targeted at graphics processing [109], Shangri-La targeted at network processing [32], and a high-level specification system for quantum chemistry computations that can generate optimized parallel code [20].

The common characteristics of all these efforts are the following: a) an understanding of application behavior at an algorithmic level, b) important properties of the microarchitecture are exposed to software layers, c) concurrency and other program properties are expressed through the language level so the compiler or hardware is not overly burdened.

While not related to these domain specific compilation and language approaches, the compilation strategy for the Cell processor shows some of these characteristics and has successfully employed techniques like compiler-supported branch prediction, compiler-assisted instruction fetch, generation of scalar codes on SIMD units, automatic generation of SIMD codes, and data and code partitioning across the multiple processor core to generate high quality code [46]. With growing heterogeneous application needs and the increasing capability and complexity of processors, we believe the lessons of such compiler and languages efforts will grow in importance and must be used to address the software complexity challenge.

9.2.3 Technology Constraints

This dissertation has focused on evaluating the performance of polymorphism and the TRIPS architecture. Other technology constraints include area, power, and increasingly reliability. We have not quantitatively addressed comparisons to other design with respect to those constraints. Clearly, a specialized processor will be more area and power efficient, but how much better compared to a polymorphous processor is not clear. Building application specific techniques for reliability are likely to make specialized processor more reliable than programmable processors. Studying polymorphism from a power, area, and reliability perspective is an exciting area of research coupled with the software complexity issues.

9.3 Final Thoughts

Polymorphism is a natural design convergence solution for future architectures that must provide massive computation power and support for heterogeneous computation needs. A partitioned design lends itself naturally to sub-division for different granularities of parallelism. The TRIPS approach of building a scalable and modular microarchitecture with concurrency expressed explicitly in the ISA is a promising direction for future architectures.

This dissertation opens up two broad areas of future work:

1. **Compiling for polymorphism:** Exposing microarchitecture-specific polymorphism techniques to the compiler introduces several challenges: 1) which microarchitecture mechanisms to expose to the software layer, 2) how to ex-

pose these mechanisms, 3) how to determine and classify program behavior, and 4) how to automatically map program behavior to the hardware mechanisms.

2. **Polymorphism to achieve other technology objectives:** While we have focused on polymorphism to improve performance, the principles of polymorphism we developed can be used for other objectives like: 1) achieving different levels of power efficiency as dictated by the environment or application, 2) providing graceful degradation of performance, and 3) improving reliability. In a more general sense, a comprehensive analysis of polymorphism with respect to all technology constraints will strengthen the case for polymorphous architectures.

In this dissertation, we developed and evaluated the idea of polymorphism and proposed a set of mechanisms targeted at supporting all granularities of parallelism - ILP, TLP, and DLP. A direct application of the ideas in this dissertation is to use these mechanisms to build a homogeneous processor that supports all granularities of parallelism. However, when a specific set of applications are of primary interest, the principles and the application classification proposed here can be used to determine which mechanisms are required to support that specific set of applications. The flexibility provided by implementing all mechanisms can be sacrificed for simplicity by implementing a subset of the mechanisms by matching the mechanisms to the application attributes.

The polymorphism framework presented here could be useful as an analysis tool while building specialized heterogeneous architectures as well. Even if a designer chooses to build some number of specialized cores, starting with polymorphous building blocks for constructing each core can help simplify the design process. Such a design choice comes about for all three granularities of parallelism. For example, to build a specialized server processor targeted primarily at TLP, the high-bandwidth memory channels and the software managed cache can be completely removed. To build a specialized processor for scientific computing that exhibits only a subset of DLP behavior, the support for MIMD execution and other specialized resources like the next-block predictor tuned for ILP can be removed.

The applications heterogeneity challenge, fundamental limitations that plague the scaling of conventional microarchitectures, and the technology limitations of power, wire-delay, and process variation present significant challenges to the performance growth curve the processor community has grown accustomed to. Architectural polymorphism, ISAs with block atomic execution with dependences explicitly encoded in them, and the principles of tiled design with well defined microarchitectural networks proposed in this dissertation provide a promising solution. We foresee several of these elements in microprocessors of the future.

Appendices

Appendix A

tsim-proc and GPA simulator comparison

In this dissertation we used two simulators for our performance evaluation. One is *tsim-proc*, which is a detailed cycle-level simulator that models the TRIPS processor at a much more detailed level than higher-level simulators like SimpleScalar [30]. Our performance validation effort showed that performance results from *tsim-proc* were on average within 10% of those obtained from the RTL-level simulator, across a large number of hand-crafted and randomly generated test programs. Because this simulator models the hardware at such a detailed level, it is not very extensible and we used a second more abstract simulator called the GPA simulator for our DLP study in chapter 8. The GPA simulator uses binaries generated by the Trimaran IMPACT compiler [162], translates instruction into a TRIPS-like instruction format and uses a scheduler that has similar heuristics to the TRIPS scheduler. In this section, we compare these two simulators and describe the differences between them.

The quantitative conclusion of this study is that the GPA simulator in the worst case over-estimates performance by 3X compared to the validated TRIPS simulator and is on average within 2X of this validated simulator. The poor code quality from the TRIPS compiler and the abstraction errors contribute roughly in equal measure to this over-estimation.

A.1 Description

The main differences between the two simulators include:

1. **ISA:** The GPA simulator uses the IMPACT compiler whose instructions are different from the TRIPS ISA. Specifically the implementation of predication in IMPACT which includes generation of complementary predicates and use of wired operators [168], is much different from the simple implementation in TRIPS. Consequently, the instruction count on TRIPS is typically higher.

2. **Compiler quality:** The IMPACT compiler is a sophisticated and heavily tuned compiler and we believe it generates higher quality code than our current TRIPS compiler. Instruction counts generated by this compiler are sometimes a factor of two less than the TRIPS compiler.

3. **Control flow:** The control flow implementation in the GPA simulator assumes multiple branches can be executed and infers that the *first* branch in serial order is the taken branch and the architecture change affected by instructions beyond it are cancelled out. Since this is a high level simulator we do not model the exact mechanisms by which this happens. In the TRIPS simulator however, explicit `null` instructions are generated for cancelling out such execution and all branches are predicated, such that during program execution exactly one branch instruction's predicate is enabled.

4. **Operand network:** The TRIPS simulator models the exact operand network protocol by modeling the control-packet and data-packet protocols of the network. The GPA simulator simply has a communication delay for operands between hops and an abstract model of a router. While this models routing contention, it does not take into account all sources of congestion in the network created by separate data and control packets.

5. **Fetch, commit, and flush networks:** The GPA simulator does not model the fetch, commit, and flush networks and instead uses fixed delays to model their behavior.

6. **Memory system:** The GPA simulator simulates the distributed data tiles and the LSQ logic by modeling 4 ports in a centralized cache which are all equidistant from the left edge of the processor core. As a result, only the horizontal routing delays are accounted for. In the case of all the loads in a program going to one single data tile, the GPA simulator ends up simulating a data tile with 4 ports and 4 operand network links.

To summarize, the GPA simulator models some microarchitecture blocks at a high level of abstraction which could result in over estimating the performance. Secondly. the richer ISA used by the IMPACT compiler allows it to generate more compact code than the TRIPS compiler which contributes to this over-estimation as well.

Benchmark					**Ratio**	
	GPA simulator		**TRIPS simulator**		**Cycles**	**Insts**
	Cycles	**Insts**	**Cycles**	**Insts**	**(T/G)**	**(T/G)**
dct	41104	148544	77998	241884	1.9	1.6
convert	29136	168000	84065	318566	2.9	1.9
highpassfilter	701236	2894135	1136573	4706789	1.6	1.6
fft	17484	33252	28501	42881	1.6	1.3
blowfish	651200	1541823	1266622	1388386	1.9	0.9
vertexsimplelight	311436	458867	844069	1010413	2.7	2.2
vertexreflection	215740	731880	538051	749745	2.5	1.0
vertexskinning	592804	1979365	1687015	2084255	2.8	1.1
fragmentsimplelight	289536	487080	581488	597852	2.0	1.2
fragmentreflection	289536	487080	400495	636232	1.4	1.3
Arithmetic Mean	313921	893002	664487	1177700	2.1	1.4

Table A.1: Comparison of GPA simulator to TRIPS simulator on the DLP kernels

A.2 Results

Table A.1 shows the comparison of the two simulators on the DLP kernels used in the DLP study in chapter 8. They were compiled using the TRIPS compiler for the TRIPS simulator and the Trimaran IMPACT compiler for the GPA simulator. The cycles and instruction counts for each simulator are shown and the last two columns show the ratio of cycles and ratio of instructions of the TRIPS simulator to the GPA simulator. The notation T/G donates ratio of TRIPS to GPA.

The GPA simulator over-estimates performance by anywhere between 1.4X to 2.9X, and on average over-estimates performance by 2X compared to the TRIPS simulator. Some of this performance difference is a result of ISA and compiler difference which is explained by the difference in instruction counts–the TRIPS simulator generates on average 1.4X more instructions. The remainder of the performance difference is a result of the abstraction errors in the GPA simulator.

To tease out the contributions from the compiler and contributions from the modeling abstractions, we simulated a suite of heavily hand optimized kernels extracted from the SPEC CPU2000 suite. Table A.2 shows the comparison of the two simulators on these kernels. For the GPU simulators these kernels were compiled using the the Trimaran IMPACT compiler, whereas for the TRIPS simulator these binaries were heavily hand optimized starting from compiler generated code.

The hand optimization reduces instruction count significantly–on average the TRIPS instruction count is 0.9 times the Trimaran instruction count, whereas on compiler generated code it was 1.4X. In fact only 2 kernels have larger instruction counts: *gzip_2* and *ammp_2*. Using such optimized code–which likely matches the code quality generated by the Trimaran compiler for the GPA simulator–creates a situation where the difference between the two simulation environments is primarily microarchitecture modeling. In this environment comparing optimized kernels, on average, the GPA simulator over-estimates performance by 1.4X.

The results from these two controlled experiments lead us to conclude that the compiler quality and the modeling errors contribute roughly in equal measure to the over estimation in performance. However, this over-estimation does not detract from the conclusions of the DLP study which uses the GPA simulation environment.

Benchmark					Ratio	
	GPA simulator		TRIPS simulator		Cycles	Insts
	Cycles	Insts	Cycles	Insts	(T/G)	(T/G)
art_2	110838	564393	72692	305790	0.7	0.5
ammp_1	184384	745950	121191	491480	0.7	0.7
equake_1	181283	939792	120943	301000	0.7	0.3
art_3	135720	615113	115014	450156	0.8	0.7
bzip2_3	234920	1133516	200774	671170	0.9	0.6
vadd	77919	590580	93625	464162	1.2	0.8
twolf_3	253946	284692	320662	289690	1.3	1.0
ammp_2	150922	515482	191693	627234	1.3	1.2
gzip_1	19915	54433	25498	17421	1.3	0.3
gzip_2	21788	51437	29998	123276	1.4	2.4
bzip2_2	176646	1019024	253706	349229	1.4	0.3
bzip2_1	213275	993654	333199	557077	1.6	0.6
art_1	39241	274744	62787	274930	1.6	1.0
sieve	150741	582570	299663	336316	2.0	0.6
parser_1	59047	258969	135733	179845	2.3	0.7
Arithmetic Mean	158133	610902	272119	516530	1.4	0.9

Table A.2: Comparison of GPA simulator to TRIPS simulator on a set of hand optimized SPEC CPU2000 microbenchmakrs

Appendix B

IPC reduction from speculation depth

This appendix contains a performance comparison of the ILP-mode of the TRIPS processor to the 1-Thread TLP configuration, where a single program is run in the TLP-mode of the processor. As a result, the speculation depth of the program is reduced and it gets to utilize only 256 of the 1024 reservation stations. This study can also be viewed as a comparison of performance from 8-deep speculation and 2-deep speculation, where speculation depth is measured in terms of number of blocks predicted.

Benchmark	**IPC**		**Slowdown**
	ILP-mode	**1-Thread**	
int/254.gap	0.9	1.4	-65.0
fp/200.sixtrack	0.9	1.5	-59.8
fp/301.apsi	2.3	2.7	-15.7
int/186.crafty	0.9	1.0	-10.2
fp/177.mesa	2.0	1.6	17.5
int/300.twolf	0.8	0.6	25.3
int/181.mcf	1.9	1.4	25.7
int/175.vpr	1.1	0.7	39.6
int/164.gzip	1.6	0.9	40.7
int/255.vortex	0.9	0.4	50.5
int/197.parser	1.0	0.5	53.4
fp/179.art	2.2	1.0	54.7
fp/168.wupwise	1.9	0.8	55.8
int/256.bzip2	1.5	0.5	66.2
fp/188.ammp	1.0	0.2	79.7
fp/183.equake	1.4	0.3	80.4
fp/171.swim	1.8	0.3	85.3
fp/172.mgrid	3.2	0.3	91.3
fp/173.applu	2.1	0.1	94.8

Table B.1: IPC comparison of ILP-mode and 1-Thread TLP-mode - SPEC CPU2000 suite.

Benchmark	IPC		Slowdown
	ILP-mode	1-Thread	
automotive/pntrch01	0.8	0.8	8.7
automotive/cacheb01	0.7	0.6	10.9
automotive/matrix01	1.0	0.9	12.9
automotive/aiifft01	1.3	1.1	15.3
networking/routelookup	0.9	0.8	15.6
office/rotate01	1.4	1.2	17.0
telecom/viterb00	1.2	1.0	17.4
automotive/puwmod01	0.9	0.7	17.8
automotive/aifftr01	1.3	1.1	18.0
automotive/ttsprk01	0.9	0.7	19.1
automotive/canrdr01	0.9	0.7	20.4
consumer/djpeg	1.3	1.0	22.7
automotive/iirflt01	0.7	0.5	25.4
automotive/rspeed01	0.9	0.7	26.9
automotive/tblook01	0.6	0.4	27.1
office/text01	1.1	0.8	27.7
networking/ospf	1.0	0.7	29.0
automotive/aifirf01	0.6	0.4	32.2
automotive/basefp01	0.6	0.4	33.6
office/dither01	1.8	1.2	33.7
consumer/cjpeg	1.6	1.0	33.7
automotive/a2time01	0.5	0.3	35.4
automotive/bitmnp01	1.3	0.8	36.7
networking/pktflow	1.2	0.7	36.7
telecom/autocor00	0.5	0.3	36.8
automotive/idctrn01	1.4	0.8	39.8
office/bezier02	1.2	0.7	41.0
telecom/fbital00	1.6	0.9	45.2
telecom/conven00	1.8	0.8	54.0
telecom/fft00	2.9	1.1	61.3

Table B.2: IPC comparison of ILP-mode and 1-Thread TLP-mode - EEMBC suite

Benchmark	**IPC**		**Slowdown**
	ILP-mode	**1-Thread**	
scientific/LU	0.7	1.3	-83.2
network/rijndael	0.3	0.3	9.0
network/blowfish	1.2	0.7	38.5
scientific/fft	1.4	0.7	51.4
graphics/fragmentreflection	1.8	0.9	51.6
graphics/vertexsimplelight	2.4	1.1	54.3
eembc/dct	4.3	1.8	58.1
graphics/fragmentsimplelight	2.4	1.0	58.6
graphics/vertexreflection	2.7	1.1	61.3
graphics/vertexskinning	4.1	1.4	65.6
eembc/highpassfilter	6.9	2.1	70.3
network/md5	0.8	0.2	70.7
eembc/convert	6.0	1.4	76.9

Table B.3: IPC comparison of ILP-mode and 1-Thread TLP-mode - DLP suite

Bibliography

[1] Alpha Architecture Handbook, Version 3, October 1996.

[2] GPGPU: www.gpgpu.org.

[3] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[4] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate Vs. IPC : The End of the Road for Conventional Microprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[5] K. Akeley. Reality Engine Graphics. In *Proceedings of the 20th Annual Conference on Computer Graphics*, pages 109–116, June 1993.

[6] D. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 248–259, Dec. 1999.

[7] D. H. Albonesi, R. Balasubramonian, S. G. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro,

P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 36(12):49–58, 2003.

[8] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proceedings of the 4th International Conference on Supercomputing*, pages 1–6, 1990.

[9] J. Andrews and N. Baker. Xbox 360 System Architecture. *IEEE Micro*, 26(2):25–37, 2006.

[10] M. Annaratone, E. A. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu, and J. A. Webb. The Warp Computer: Architecture, Implementation, and Performance. *IEEE Transactions on Computers*, 36(4):1523–1538, December 1987.

[11] Arvind and D. E. Culler. Dataflow Architectures. *Annual Review of Computer Science*, 1:225–253, 1986.

[12] Arvind and K. Gostelow. The U-Interpreter. *Computer*, 15(2):42–49, 1982.

[13] Arvind and R. S. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[14] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 218–229, 2001.

[15] S. Balakrishnan and G. S. Sohi. Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 302–313, June 2006.

[16] M. Baron. MP Cores for Handheld Apps. *Microprocessor Report*, 19(12), December 2005.

[17] M. Baron. OMAP3 Sets Specs for Cellphones. *Microprocessor Report*, 20(4), April 2006.

[18] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a Scalable Architecture based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[19] V. Baumgarte, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP – A Self-Reconfigurable Data Processing Architecture. In *1st International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2001.

[20] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models. *Proceedings of the IEEE*, 93(2):276–292, 2005.

[21] T. Blank. The Maspar MP-1 architecture. In *Proceedings of the IEEE Compcon, Spring 1990*, pages 20–24.

[22] D. W. Blevins, E. W. Davis, R. A. Heaton, and J. H. Reif. BLITZEN: A Highly Integrated Massively Parallel Machine. *Journal of Parallel and Distributed Computing*, 8(2):150–160, 1990.

[23] J. Blow. Game Development: Harder Than You Think. *ACM Queue*, 1(10), February 2004.

[24] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of Supercomputing 1988*, pages 330–339, November 1988.

[25] S. Y. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, 2005.

[26] V. Bove and J. Watlington. Cheops: A reconfigurable data-flow system for video processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(2):140–149, 1995.

[27] I. Buck. Data parallel computation on graphics hardware. In *Graphics Hardware 2003: Panel Presentation*, 2003.

[28] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[29] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial Computation. In *Proceedings of the 11th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 14–26, October 2004.

[30] D. Burger and T. M. Austin. The SimpleScalar Tool Set Version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[31] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[32] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju. Shangri-La: Achieving High Performance from Compiled Network Applications while Enabling Ease of Programming. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 224–236. ACM Press, 2005.

[33] The Connection Machine CM-2 Technical Summary, April 1987.

[34] K. Coons, X. Chen, S. Kushwaha, K. S. McKinley, and D. Burger. A Spatial Path Scheduling Algorithm for EDGE Architectures. In *Proceedings of the 12th International Conference on Architectural Support for Programmin Languages and Operating Systems*, October 2006.

[35] B. Copeland. Colossus: its origins and originators. *IEEE Annals of Computing*, 26:38–45.

[36] N. Corp. NVIDIA GPU programming guide, v2.2.1, November, 2004.

[37] A. Cristal, O. J. Santana, M. Valero, and J. F. Martinez. Toward kilo-instruction processors. *ACM Transactions on Architecture and Code Optimization*, 1(4):389–417, December 2004.

[38] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, April 1991.

[39] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labont, J. H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with Streams. In *The Proceeding of the 2003 International Conference for High Performance Computing, Networking, Storage, and Analysis*, November 2003.

[40] J. Dennis and D. Misunas. A preliminary architecture for a basic dataflow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, pages 126–132, January 1975.

[41] R. Desikan. *Distributed Selective Re-Execution for EDGE Architectures*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, December 2005.

[42] R. Desikan, D. Burger, and S. W. Keckler. Measuring Experimental Error in Microprocessor Simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, July 2001.

[43] K. Diefendorff and P. K. Dubey. How Multimedia Workloads Will Change Processor Design. *Computer*, 30(9):43–45, 1997.

[44] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, 2000.

[45] P. Dubey. Recognition, Mining and Synthesis Moves Computers to the Era of Tera. *Intel Technology Magazine*, February 2005.

[46] A. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. O. D. Prener, J, Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing Compiler for the Cell Processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 66–76, September 2005.

[47] Embedded Microprocessor Benchmark Consortium. *EEMBC*, 2000.

[48] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. thew Mattina, and A. Seznec. Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings of The 29th International Symposium on Computer Architecture*, pages 281–292, May 2002.

[49] R. Espasa, M. Valero, and J. E. Smith. Out-of-Order Vector Architectures. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 160–170, December 1997.

[50] J. Eyre and J. Bier. DSP processors hit the mainstream. *IEEE Computer*, 31(8):51–59, 1998.

[51] R. Fernando and M. J. Kilgard. *The Cg Tutorial*. Addison-Wesley Publishing Company, 2003.

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

[53] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, June 1995.

[54] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: Letting applications define architectures. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 324–335, December 1996.

[55] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transaction on Computers*, 21(C):948–960, 1972.

[56] D. Folegnani and A. González. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 230–239, June 2001.

[57] M. Frigo. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 169–180. ACM Press, 1999.

[58] P. N. Glaskowsky. PACT Debuts Extreme Processor. *Microprocessor Report*, 14(10), October 2000.

[59] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, 33(4):70–77, April 2000.

[60] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, October 2002.

[61] P. Gratz, C. Kim, R. McDonald, S. W. Keckler, and D. Burger. Implementation and Evaluation of On-Chip Network Architectures. In *Proceedings of the 24th International Conference on Computer Design*, pages 170–177, October 2006.

[62] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, 2001.

[63] R. Gupta. A fine-grained MIMD architecture based upon register channels. In *Proceedings of the 23rd Annual Workshop and Symposium on Microprogramming and Microarchitecture*, pages 28–37, 1990.

[64] S. Guyer and C. Lin. Broadway: a compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, 2005.

[65] Z. S. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 108–120, June 1997.

[66] T. R. Halfhill. PicoChip makes a Big MAC. *Microprocessor Report*, 17(10):17–19, October 2003.

[67] T. R. Halfhill. ClearSpeed Hits Design Targets. *Microprocessor Report*, 18(1):16–17, January 2004.

[68] T. R. Halfhill. Busy bees at Silicon Hive. *Microprocessor Report*, 19(6):17–20, June 2005.

[69] T. R. Halfill. MathStar Challenges FPGAs. *Microprocessor Report*, 20(7):29–35, July 2006.

[70] L. Hammond, B. A. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, 1997.

[71] R. W. Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *ASP-DAC*, pages 564–570, 2001.

[72] R. W. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE*, pages 642–649, 2001.

[73] A. Hartstein and T. R. Puzak. The optimum pipeline depth for a microprocessor. In *Proceedings of The 29th Annual International Symposium on Computer Architecture*, pages 7–13, June 2002.

[74] A. Hartstein and T. R. Puzak. Optimum power/performance pipeline depth. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 117–128, December 2003.

[75] J. Hauser. The SoftFloat and TestFloat Packages, http://www. jhauser.us/ arithmetic/ index.html.

[76] J. R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 16–18, April 1997.

[77] J. Heinrich. MIPS RISC Architecture, Volume I: Introduction to the ISA (2nd ed.). Document Number 007-3515-001/007-3576-001, Feb 5, 1998.

[78] J. Hennesy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.

[79] H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the 11th International Conference on High-Performance Computer Architecture*, pages 258–262, February 2005.

[80] M. S. Hrishikesh, D. Burger, S. W. Keckler, P. Shivakumar, N. P. Jouppi, and K. I. Farkas. The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays. In *Proceedings of The 29th International Symposium on Computer Architecture*, pages 14–24, June 2002.

[81] H.T. Kung and C.E. Leiserson. Systolic arrays (for VLSI). In Sparse Matrix Proceedings, 1979.

[82] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control Flow Speculation in Multiscalar Processors. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 218–229, Feb. 1997.

[83] R. M. Jenevein and J. C. Browne. A control processor for a reconfigurable array computer. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 81–89, 1982.

[84] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), September 2005.

[85] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient Conditional Operations for Data-parallel Architectures. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 159–170, December 2000.

[86] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles. Stream Scheduling. In *Proceedings of the 3rd Workshop on Media and Streaming Processors*, pages 101–106, December 2001.

[87] S. W. Keckler and W. J. Dally. Processor coupling: integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202–213. ACM Press, June 1992.

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

[89] C. Kim, D. Burger, and S. 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 Programmin Languages and Operating Systems*, pages 211–222, October 2002.

[90] H.-S. Kim and J. E. Smith. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 71–80, June 2002.

[91] A. KleinOsowski and D. J. Lilja. Computer Architecture Letters, Volume 1, June, 2002.

[92] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[93] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick. Vector IRAM: A Media-oriented Vector Processor with Embedded DRAM. In *12th Hot Chips Conference*, August 2000.

[94] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 399–409, June 2003.

[95] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The Vector-Thread Architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 52–63, June 2004.

[96] K. Krewell. IDF Delivers Extreme Surprises. *Microprocessor Report*, 17(10):7–8, October 2003.

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

[98] K. Krewell. Startup Aegia Accelerate Reality. *Microprocessor Report*, 19(4), April 2005.

[99] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 81–92, June 2003.

[100] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *MICRO*, pages 195–206, 2004.

[101] A. Kunimatsu et. al. Vector Unit Architecture For Emotion Synthesis. *IEEE Micro*, 20(2):40–47, March 2000.

[102] L.E. Shar and E.S. Davidson. A Multiminiprocessor System Implemented Through Pipelining. *IEEE Computer*, 7:42–51.

[103] R. B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, 1996.

[104] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. 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, New York, NY, USA, 1998. ACM Press.

[105] B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging Head and Tail Duplication for Convergent Hyperblock Formation. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, December 2006.

[106] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, June 1992.

[107] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 161–171, June 2000.

[108] W. R. Mark and D. Fussell. Real-time rendering systems in 2010. Technical Report TR-05-18, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, May 2005.

[109] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Proceedings of the 30th Annual Conference on Computer Graphics*, 2003.

[110] R. McDonald, R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. TRIPS Instruction Set Architecture (ISA) Manual. Technical Report TR-05-19, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, May 2005.

[111] K. S. McKinley, J. Burrill, B. Cahoon, J. E. B. Moss, Z. Wang, and C. Weems. The Scale Compiler. Technical report, University of Massachusetts, 2001. http://ali-www.cs.umass.edu/∼scale/.

[112] B. Moore, A. Padegs, R. Smith, and W. Bucholz. Concepts of the System/370 Architecture. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 282–292, June 1987.

[113] C. R. Moore. Managing the Transition from Complexity to Elegance: Design Convergence. *IEEE Micro*, 24(1):79–80, 2004.

[114] R. Nagarajan. *Design and Analysis of Technology Scalable Architectures, draft version, December 2006*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences.

[115] R. Nagarajan, X. Chen, R. G. McDonald, D. Burger, and S. W. Keckler. Critical Path Analysis of the TRIPS Architecture. In *Proceedings of the IEEE International Symposium on Performance Analysis (ISPASS)*, pages 37–47, March 2006.

[116] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. W. Keckler. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In *13th International Conference on Parallel Architecture and Compilation Techniques*, pages 74–84, October 2004.

[117] R. Nagarajan, K. Sankaralingam, S. W. Keckler, and D. Burger. A Design Space Evaluation of Grid Processor Architectures. In *Proceedings of the*

34th Annual International Symposium on Microarchitecture, pages 40–51, December 2001.

[118] J. R. Nickolls and J. Reusch. Autonomous SIMD flexibility in the MP-1 and MP-2. In *SPAA '93: Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 98–99, New York, NY, USA, 1993. ACM Press.

[119] P. S. Oberoi and G. S. Sohi. Parallelism in the Front-End. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 230–240, June 2003.

[120] J. Oliver, R. Rao, P. Sultana, J. Crandall, E. Czernikowski, L. J. IV, D. Franklin, V. Akella, and F. T. Chong. Synchroscalar: A Multiple Clock Domain, Power-Aware, Tile-Based Embedded Processor. In *Proceedings of the 30th Annual International Symposium on Architecture*, pages 150–161, June 2004.

[121] T. Olson. Advanced Processing Techniques Using the Intrinsity FastMATH Processor. In *Embedded Processor Forum*, May 2002.

[122] A. Pajuelo, A. Gonzalez, and M. Valero. Speculative Dynamic Vectorization. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 271–280, May 2002.

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

[124] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[125] 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 *IEEE International Solid-State Circuits Symposium*, February 2005.

[126] M. Pharr and G. Humphreys. *Design and Implementation of a Physically-Based Rendering System*. Morgan Kaufmann, 2003.

[127] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 90–101, June 2001.

[128] M. Pschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE special issue on Program Generation, Optimization, and Adaptation*, (2):232–275, 2005.

[129] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A Scalable Instruction Queue Design Using Dependence Chains. In *Proceedings of the 29th Annual*

International Symposium on Computer Architecture, pages 318–327, June 2002.

[130] R. Rabbah, I. Bratt, K. Asanovic, and A. Agarwal. Versatility and VersaBench: A New Metric and a Benchmark Suite for Flexible Architectures. Technical Report MIT-LCS-TM-646), Massachusetts Institute of Technology, June 2004.

[131] S. Rajagopal, S. Rixner, and J. Cavallaro. A programmable baseband processor design for software defined radios. In *Proceedings of the IEEE International Midwest Symposium on Circuits and Systems*, pages 413–416, 2002.

[132] N. Ranganathan and M. Franklin. An empirical study of decentralized ILP execution models. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–281, October 1998.

[133] N. Ranganathan, R. Nagarajan, D. Burger, and S. W. Keckler. Combining hyperblocks and exit prediction to increase front-end bandwidth and performance. Technical Report TR-02-41, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, September 2002.

[134] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1):9–50, 1993.

[135] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3–13, December 1998.

[136] Roddy Urquhart and Will Moore and Andrew McCabe. *Systolic Arrays*. Institute of Physics Publishing, 1987.

[137] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, December 1997.

[138] R. M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 22(1):64–72, January 1978.

[139] R. Saasnka. *ALP: Energy Efficient Support for All Levels of Parallelism for Complex Media Applications*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Sciences, July 2005.

[140] K. Sankaralingam, R. Nagarajan, D. Burger, and S. W. Keckler. A Technology Scalable Architecture for Fast Clocks and High ILP. In *Proceedings of the 5th Workshop on the Interaction of Compilers and Computer Architecture*, January 2001.

[141] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, S. W. Keckler, D. Burger, and C. R. Moore. Exploiting ILP, TLP and DLP with the Poly-

morphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[142] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, December 2006.

[143] K. Sankaralingam, V. A. Singh, S. W. Keckler, and D. C. Burger. Routed Inter-ALU Networks for ILP Scalability and Performance. In *Proceedings of the 21st International Conference on Computer Design*, pages 170–177, October 2003.

[144] M. C. Sejnowski and et al. Overview of the Texas Reconfigurable Array Computer. In *AFIPS Conference Proceedings*, pages 631–642, 1980.

[145] A. Seznec and R. Espasa. Conflict-free accesses to strided vectors on a banked cache. *IEEE Transactions on Computers*, 54(7):913–916, 2005.

[146] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *Fourth International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, pages 185–189, March 2006.

[147] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow Predication. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, December 2006.

[148] B. Smith. Architecture and applications of the HEP multiprocessor computer system. In SPIE Real Time Signal Processing IV, pages 241–248, 1981.

[149] J. E. Smith, G. Faanes, and R. A. Sugumar. Vector instruction set support for conditional operations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 260–269, June 2000.

[150] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[151] E. Sprangle and D. Carmean. Increasing Processor Performance by Implementing Deeper Pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 25–36, June 2002.

[152] S. Srinivasan, R. Rajwar, H. Akkary, A. Ghandi, and M. Upson. Continual flow pipelines. In *Proceedings of the 11th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, October 2004.

[153] Standard Performance Evaluation Corporation. *SPEC CPU2000*, 2000.

[154] O. Takahashi, S. Cottier, S. H. Dhong, B. Flachs, and J. Silberman. Power-Conscious Design of the Cell Processor's Synergistic Processor Element. *IEEE Micro*, 25(5):10–18, 2005.

[155] D. Talla, L. John, and D. Burger. Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements. *IEEE Transactions on Computers*, pages 35–46, 2003.

[156] M. B. Taylor, J. Kim, J. Miller, D. W. laff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, W. L. Jae-Wook Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The RAW Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, March 2002.

[157] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 341–353, February 2003.

[158] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. P. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 2–13, 2004.

[159] J. M. Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 26(1):5–26, January 2001.

[160] J. E. Thornton. Parallel Operation in the Control Data 6600, pp. 33-41 in AFIPS Conference Proceedings, 1964 Fall Joint Computer Conference, Spartan Books Inc., Washington D.C. (1965).

[161] M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10–20, 1996.

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

[163] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, New York, NY, USA, June 1996. ACM Press.

[164] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22rd Annual International Symposium on Computer Architecture*, pages 191–202, June 1995.

[165] J. Turley. Tensilica CPU Bends to Designers's Will. *Microprocessor Report*, 13(4), March 1999.

[166] T. Ungerer, B. Robi, and J. Silc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.

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

[168] V.Kathail, M.Schlansker, and B.R.Rau. HPL-PD Architecture Specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, February 2000.

[169] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: RAW Machines. *Computer*, 30(9):86–93, 1997.

[170] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[171] O. Wolf and J. Bier. StarCore Launches First Architecture. *Microprocessor Report*, 12(14):17–20, October 1998.

[172] L. Wu, C. Weaver, and T. Austin. CryptoManiac: A Fast Flexible Architecture for Secure Communication. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 110–119, June 2001.

[173] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Proceedings of the 4th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 49–58, June 1995.

[174] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. M. Baas. An Asynchronous Array of Simple Processors for DSP Applications. In *Proceedings of the IEEE International Solid-State Circuits Conference, (ISSCC '06)*, pages 428–429, 2006.

Vita

Karthikeyan Sankaralingam was born in Chennai, India on 6th February 1978, the son of Parathesi Sankaralingam and Aiyasammi Sankarammal. He received the Bachelor of Technology degree in Aerospace Engineering from the Indian Institute of Technology, Madras in 1999. He entered the graduate program in Computer Sciences at the University of Texas at Austin in August 1999. He received a Master of Science degree in August 2006.

Permanent address: 1911 Willow Creek Dr.
Apt. 205
Austin, Texas 78741

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.