Polymorphism: A Unifying Approach for Managing Processor Complexity, Enabling Technology Scalability, and Extracting Concurrency

Publication No. _____

Karthikeyan Sankaralingam, Ph.D.
The University of Texas at Austin, 2006

Supervisor: Stephen W. Keckler

Processor architects today are faced by two daunting challenges, ensuring applications with heterogeneous computation needs, and certain edge 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 some design philosophy 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 such heterogeneous computation by supporting different granularities of parallelism. Polymorphism configures coarse grain microarchitecture blocks to provide an adaptive and flexible processor architecture. Technology scalability is achieved by designing an architecture using scalable and modular microarchitecture blocks.

We use the dataflow paradigm as the unifying abstraction layer across these types of parallelism. All programs are expressed in terms of dataflow graphs and directly mapped to the hardware, appropriately partitioning as required by the granularity of parallelism. We introduce EDGE ISAs - a class of ISAs, as an architectural solution for efficiently expressing parallelism for building technology scalable architectures.

We developed, the TRIPS architecture as an implementation of EDGE which uses 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. We have also built a prototype TRIPS chip in 130nm ASIC technology composed of two processor cores and a distributed 5MB non-uniform (NUCA) on-chip memory system.

Our performance results show that the TRIPS microarchitecture can run high-ipc dataflow-like applications. On a set of hand-optimized kernels IPCs in the range of 4 to 8 are seen, and on a set of highly data parallel benchmarks with compiler generated code IPCs in the range of 3 to 6 are seen. On the EEMBC and SPEC CPU2000 benchmarks we see IPCs in the range of 0.5 to 2.2. With more aggressive compiler optimization we expect these numbers to improve.

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.

Acknowledgments

"At times one sees lights go out and is shielded by a spell-inducing power. Each of us has more intimate will-o-the-wisps guiding and bewildering us from within."

-Michael Schoenfeld

Many people have contributed to my dissertation research and my experiences 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. Steve Keckler and Doug Burger provided the vision, constant encouragement and technical expertise that is manifested in the dissertation possible. Steve also was an excellent mentor and thesis advisor and I cannot thank him enough for his tireless labor and wisdom.

Doug Burger, as the co-leader of the CART group, and a dedicate research advisor during 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 problems to brewing beer.

I would like to acknowledge the moral and technical support that I have received from the students in my research group, CART. 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.

I would especially like to thank Ramadass Nagarajan with whom I had a close collaboration 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 forays in writing a PowerPC part of the SimpleScalar simulator, to our late night brain-storming discussions on initial ideas for the TRIPS processor, and design and verification of the prototype chip, I have always enjoyed Ramadass's insights and bouncing his perspective. The instructions set described in this dissertation and many aspects of the microarchitecture were jointly designed by us.

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 flexible staff in the department were outstanding, and the exam times they enjoyed is not HOPELESS is one more complexity avoided. I would like to thank Gina Nistler for helping with graduate school microlibrary, 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 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.

Chapter 1
Introduction

In the last decade, programmable processors have proliferated into increasingly diverse application domains, producing distinct markets for desktop, network, server, consumer, graphics, and digital signal processors. While clearly parallel applications - possibly performance improvements, these processors perform poorly on applications outside of their intended domains, 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 modeling of video database introduce the need for computation systems that can support such heterogeneous computation. Future systems can be heterogeneous at the hardware level and can be 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 micro architecture structures to improve performance. Due to technology limitations of wire delays [8], power [73], and process variation [31], performance improvements 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 grain data level parallelism, and not rely on only fine grain 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, memories 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 reliable and modular architectures.

Broadly, the two trends that processor architects foresee: 1) emerging applications with heterogeneous computation needs, and 2) technology limitations of power, wire-delay, and process variation. There is a growing need for designs 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. To key ideas behind polymorphism is to configure coarse grain microarchitecture blocks to provide an adaptive and flexible processor architecture. Technology scalability is achieved by designing an architecture using scalable and modular microarchitecture blocks.

Another emerging trend affecting 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 efficiently. The Tarantula processor is one example of integrated heterogeneity [35]. The two major downsides to this approach are increased hardware complexity since there is little design reuse between the types of processors and thus 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 problems. 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 economies of scale benefits are maintained across many instructions, thus saving energy per executed instruction.

To manage design complexity and address wire delay scaling, the polymorphous architecture is constructed using microarchitecture control and data networks which only connect neighboring blocks used for communication.

One contribution of this dissertation is this scalable and distributed architecture. The mechanisms to implement polymorphism are developed in the context of this architecture. We use 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. I encode direct instruction-to-instruction communication and reduce the overheads of discovering and managing dependences 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 is under often interfaces 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 a limited number of dependences are ever need the instruction space. Dependences inside such a block are encoded directly in the instructions, while dependences across blocks are supported through architectural register or memory. This execution model forms, 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 implementation core is completely distributed using microarchitecture control and data networks which only connect neighboring blocks used for communication.

1.1 Principles of Polymorphism

We define **architectural polymorphism** as the ability to modify the functionality of coarse grain 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, as best fits application-specific structures.
- Reconfiguring coarse grain blocks to provide different functionality instead of synthesizing fine grain primitive components into blocks with different functionality. Polymorphism refers to configuring microarchitecture blocks to provide different functionality and is different from FPGA-like reconfiguration.

1.2 System design questions

Before applying the abstract definition of architectural polymorphism to processor architectures to develop the measures and mechanisms for implementing polymorphous systems, three main system questions 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.

1. Ultra-fine grained FPGAs which consist of an array of gate or coarse-grain logic blocks interconnected through a configurable network. These are typically programmed using a high-level hardware description language and applications are synthesized to the hardware.
2. Lots of basic processing cores like in Picochip [58] or PACT-XPP [38]. The primitive processor elements are more powerful than gates and lookup tables like in an FPGA. Because they are programmed at a higher level of abstraction they speed up the development process. Although they are programmed at a higher level of abstraction than FPGAs they still perform poorly on applications in hardware.
3. Many simple-to-medium processors like in the RAW architecture [67, 24] or Sun Niagara chip [45]. Each processing core is a full fledged processor that may optimizations compiled down to the ISA of the processor. RAW also has the ability to use sophisticated compiler techniques to map a single application across the processing cores.
4. Many powerful-to-border processors like in the Power4 chip [68]. The processing cores are many powerful and tightly-knit performance than the above three.
5. Some number of ultra-wide-issue processors like the Grid Processor [13] or a TRIPS chip like configuration we propose in this dissertation.

The fine grain architecture performs well when ample fine grain parallelism exists but do not support general purpose sequential programs. They are plagued by synchronization overhead resulting from aggregating multiple of these units together. Coarse grain architectures using conventional wide-issue out-of-order processors have the ability for high performance on sequential code, but have traditionally lacked the capability for parallelism and support for fine grain parallelism.

In this dissertation, we assert that fewer number of large cores are better than more fine grained cores across a spectrum of applications if the coarse grain 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 for rediscovering concurrency, and reduce the hardware overheads of instruction-level scheduling. 2) The full processor core is designed to exploit coarse grain concurrency, and use polymorphism to subdivide resources to support fine grain 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 broken into three types of parallelism:

Instruction-level Parallelism (ILP): The predominant type of parallelism in among individual machine operations, such as memory load, store, arithmetic operations. The operations are simple RISC-style operations and the system is handed a single program written with a sequential processor in mind [26].

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.

The differences between application domains in the concurrency attributes which include the following:

- Memory access patterns which include streaming-like regular or more irregular access typical of memory intensive data structures.
- Instruction mix.
- Type 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 100W to 300W regime.

However, at an architecture level, granularity of parallelism is the main difference between different application domains. It should also be noted that these classes of concurrency are not mutually exclusive. In fact, it is common to extract some amount of ILP in traditional multi-threaded workloads like database workloads. An example of simultaneously using TLP and DLP is the IBM Cell processor, where multi-threading 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 7 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 memories tables, to extract concurrency [5]. Increasing wire delays and the limits on pipeline depth from a performance and power perspective restrict the scalability of these architectures [7, 71, 73, 77, 102]. Consequently, technology limitations have driven a desire for scalability, modularity, reduced complexity, and energy efficiency in processor architectures.

Polymorphism also requires some of these very same properties and achieves these goals.

- **Scalability and Modularity:** The basic ideas behind polymorphism lead to the construction of scalable and reconfigurable modular blocks to support multiple application classes.
- **Complexity:** The economy of mechanisms that is central to architectural polymorphism inherently reduces complexity and it 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 or 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 in the hardware. The hardware is implemented using a distributed microarchitecture that relies on well defined control and data networks for

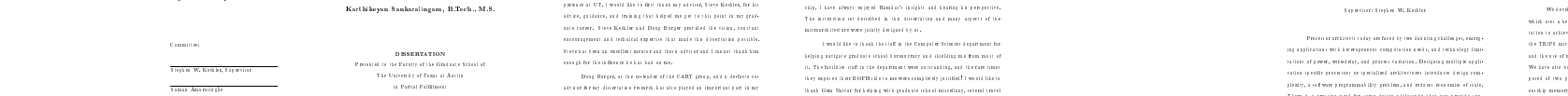

Figure 1.2: Granularity of parallel processing elements on a chip. Numbers of cores that can fit on a typical 90nm chip.

Architecture type	Processor type	Processor granularity	Configurable granularity
Programmable ASIC	Heterogeneous	Coarse grain	Coarse grain
Application specific ASIC	Heterogeneous	Fine grain	Fine grain
Polymorphous Architectures			
Polymorphic	Homogeneous (1)	Coarse grain (2)	Coarse grain
	Heterogeneous	Fine grain	

Table 1.1: A taxonomy of architectures.

core application specific hardware. Polymorphous architectures can tailor a programmable architecture to application needs.

- **Processor types:** While polymorphous does not require or imply chip multiprocessors processors or cores, in this dissertation we restrict ourselves to discussing and evaluating polymorphous for homogeneous cores. The Smart Memories chip is another example of a homogeneous polymorphous architecture.
- **Processor granularity:** Architectural polymorphism can be implemented on fine grain cores like simple in-order processors or coarse grain cores like in TRIPS architecture. Designing polymorphous mechanisms for aggregating fine grain cores to execute a large processors presents different challenges from partitioning a coarse grain core further supporting fine grain concurrency.
- **Configuration granularity:** Architectural polymorphism is defined as configuration of coarse grain microarchitecture blocks and do different from reconfiguring different functions from fine grain primitive components like datapath slices, like and FPGAs, 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 mechanisms in the TRIPS processor are: the instruction window

Application type	Processor type	Processor granularity	Configurable granularity
Programmable ASIC	Heterogeneous	Coarse grain	Coarse grain
Application specific ASIC	Heterogeneous	Fine grain	Fine grain
Polymorphous Architectures			
Polymorphic	Homogeneous (1)	Coarse grain (2)	Coarse grain
	Heterogeneous	Fine grain	

Table 1.1: A taxonomy of architectures.

space, phase coherent flow, the block sequencing logic, and the on-chip memory system.

While the concepts and mechanisms are explained in details Chapter 4 we briefly summarize the concepts and provide some examples of polymorphism below. Using polymorphism the execution stations can be reconfigured in the following ways: a single ALU in the processor to different granularities of parallelism: 1) configure the reservation stations like an instruction window and dataflow entries to one thread to support ILP, 2) share the reservation stations among multiple threads for TLP, and 3) provide instruction storage and support at every ALU site to support fine grain DLP that is best executed in a SIMD style of computation.

1.5 Thesis Statement

This dissertation introduces the concept of architectural polymorphism – the capability to configure coarse grain 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 grain 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 grain 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

[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.

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 TRIPS processor cores and a distributed 1MB non-uniform (NUCA) on-chip memory system.

Data-level 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 3.2. 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, 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. 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 rest of this dissertation is organized as follows. Chapter 2 discusses related work and places this dissertation in the context of prior work. Chapter 3 describes the TRIPS architecture and the prototype TRIPS chip. We describe the main features of EDGE ISAs, describe the microarchitecture of the TRIPS chip and briefly describe the logic design, verification, synthesis and physical design of the prototype TRIPS chip.

Chapter 4 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.

Chapter 5 presents a performance evaluation of the TRIPS processor focussed on instruction-level parallelism. The performance evaluation is based on an event driven validated processor simulator. Chapter 6 presents a performance evaluation of extracting thread-level parallelism using polymorphous

mechanisms in the TRIPS processor.

Chapter 7 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 8 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 focussed on support for different types of applications on a single substrate using reconfiguration or other means.

Multithreading: While multithread is not directly related to supporting different types of applications, polymorphism like behavior has been used to support multihreading 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 [97]. 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 [151]. 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 [139]." Other recent systems that provided multithreading support on a single VLSI chip include Hydra [68], the MIT M-Machine [50], MIT Alewife machine [3], and the Piranha multiprocessor [17].

Fine grain multithreading to share processor resources between threads has been explored using different techniques. The Tera computer system had support for fine-grain multithreading interleaving 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 [84]. Both of these in particular 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 [155]. They proposed an innovative method of replicating certain architectural storage elements in the processor, but sharing 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 [164]. Ungerer

et al. provide a detailed survey of multithreading literature [157].

Novel architectures: Browne et al. developed the Texas Reconfigurable Array Computer that could support sequential processing, SIMD, and MIMD processing on a single substrate [47, 80]. The TRAC project was focussed on building interconnection networks and optimizing communication for a configurable array that relied on a lot off-chip communication.

The Stanford Smart Memories employs project polymorphous mechanisms to synthesize a large core from a modular substrate [101]. 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 exit 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 exit 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 [91]. This architecture provides an innovative way of building a scalable, tightly integrated MIMD array for data intensive processing. Clearly this architecture can excel on vector codes and fine grain MIMD parallelism. 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 [132]. 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 is able to exploit the simple implementation of sub-word SIMD already common in many machines and it 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. The main drawback of ALP 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 DLP 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 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. introduced a *versatility* metric to quantify the ability of an architecture to effectively execute a broad set of applications [122]. They also proposed a VersaBench suite that comprised of a set of applications that captured 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. [6] 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 mechanism to provide adaptability include the following: adjusting cache size

via ways [7], sizing issue windows [54], adjusting the issue window coupled with the load/store queue and register file [119], adjusting issue width along with the functional units [14], and adaptively resizing instruction issue queues [77, 121].

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 [94]. 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 [95].

Coarse-grained Reconfigurable architectures: Fisher et al. proposed Custom-fit processors where processor cores are synthesized at design time based on application needs [51]. They adopt a unique approach of designing a heavily customizable VLIW architecture in which the number of registers, memory sizes and hierarchy, number and types of functional units 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 [156].

PACT-XPP is an innovative array based architecture for stream com-

putation which does data-flow computing in the array [18]. 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 [67] 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. Hartenstein has written a literature survey of other reconfigurable coarse grained architectures targeted at a single application domain [69, 70].

In the ASH architecture, the predication model and dataflow concepts are similar to the TRIPS approach [27]. 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 [73].

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 polymorphism 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 [75, 105, 131]. 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 [45, 46, 90, 140]. Removing these limitation 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 [89]. 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. These architec-

tures 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 [78]. Prior to this formal definition and specification of systolic arrays, the British Colossus computer employed an architecture similar to systolic arrays for code breaking [33]. In general, systolic arrays have primarily been used to build special purpose application specific hardware [129]. 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 [22]. 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 term was coined by Flynn in his taxonomy of computer architectures [53]. The early fine-grain SIMD machines like the CM-2 [31] and MasPar MP-1 [20] provided high ALU density but lacked support for fine-grain 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 [34]. 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 [24]. The Briarcliff architecture is a fine grain 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 [60]. The use of register channels in this architecture is similar to the uses of FIFOs used in the Instruction Level Distributed Processing architecture [87]. The most prevalent use of fine grain MIMD processing is in modern graphics processors which contain vertex shaders that are MIMD architectures [102, 103].

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 dubbed a stream processor, is a SIMD/vector hybrid using a SIMD control unit coupled with a memory system resembling a vector machine [127]. 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 [37, 57]. The Brook programming language provides support for stream computation on graphics hardware [26].

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 [45]. 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 [136]. 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 [114]. This architecture is also heterogeneous since it provides two dedicated datapaths.

Intrinsity is an embedded processor that includes a high performance scalar MIPS32 core integrated with an array based parallel vector math unit [113]. 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 2Ghz. The instruction control in the array is strict SIMD with each ALU executing the same instruction every cycle. The IBM Cell processor is another example of a hybrid architecture that includes an out-of-order processor and up to eight SIMD processors, dubbed *synergistic processor engines*, with a software managed memory system [76, 81, 117]. The out-of-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 [99, 147–149, 160]. In the RAW architecture, all tiles are identical and include a processor core, a router, memory ordering logic, and data stor-

age which is configured as a data cache. The Piranha architecture explored tiled architectures targeted at server workloads and took an extreme position, for the time [17]. 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 [101]. The Synchroscalar [112] and AsAP [165] architectures are other examples of homogeneous tiled architecture which are less general and specifically targeted at DSP applications. Emerging fine-grained CMP architectures, such as Sun's Niagara [88, 92] or IBM's Cell [81], can also be viewed as tiled architectures. Other examples of tiled architectures targeted at specific domains include Starcore [162], Picochip [64], Clearspeed [65], and Silicon Hive [66] many of which are reviewed here [63].

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 conlcude 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 [38]. 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 [13]. Continuing this work on dynamic dataflow Arvind and Nikhil proposed a Tagged-Token Dataflow architecture with purely data-driven instruction scheduling for programs expressed in a dataflow language [12]. 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 [36]. The rich history of dataflow architectures is review 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: Technology limits of power, design complexity, and wire delays are the constraints that have led architects to build scalable and modular designs. Processor performance growth in the future, atleast 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 than 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 [124]. Sohi et al. proposed Multiscalar processors, in which a single program is broken up into a collection of speculative tasks [141]. A different approach to creating a distributed window uses dynamic traces for the execution partitions [158]. 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 [87].

Other current research efforts targeting ILP are focused on large-window parallelism by means of checkpointing and speculation [35, 143], hybrid dataflow speculation [15], and out-of-order processor frontend microarchitecture mech-

anisms [111]. 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 [107].

Chapter 3

TRIPS Architecture

The TRIPS architecture defines a new instruction set architecture called Explicit Data Graph Execution (EDGE) which allows dependences to be explicitly encoded in the instructions. The microarchitecture is heavily partitioned and uses well defined communication networks to build a large, coarse-grained processors (also known as Grid Processors) to achieve high performance on single-threaded applications with high ILP. These cores are augmented with polymorphous features that enable the compiler or run-time system to subdivide the core for explicitly concurrent applications at different granularities. Contrary to conventional large-core designs, with centralized components that are difficult to scale, the TRIPS architecture is heavily partitioned to avoid these large centralized 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 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 appro-

priate 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 both software and hardware complexity and configuration overheads. The remainder of this chapter first describes EDGE ISAs and the execution model for such ISAs. We then describe the TRIPS instruction set which is one instantiation of EDGE ISAs, the TRIPS processor microarchitecture, an overview of polymorphism, and a description of the prototype TRIPS chip.

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 direct 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. 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. The dataflow graph abstraction amortizes the overheads of instruction management across several instruction in a full block of instructions. 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, during register renaming, the overheads of instruction management affect the scalability of hardware and limit achievable performance. The block atomicity, amortizes these overheads across many instructions and expresses concurrency efficiently to the hardware.

Technology Scalability: EDGE ISAs amortize per-instruction bookkeeping over a large number of instructions and reduce the number of branch predictions and register file accesses. Encoding dependences explicitly in the instructions simplifies dependence checking hardware. 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 [125].

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 [100], or a run-time trace [130].

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 sub-

sequent blocks, and (3) *block temporaries*, which are values that are produced and consumed 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 block outputs of one block and the block inputs of its successor 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 a 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 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 [12, 38]. 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 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 grid. The experimental results in Chapter 5 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 [110] convinced us of the potential of this architecture and execution model. In that study, we used the Trimaran compiler infrastructure [153] 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 existing 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. the grid. 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 [115]. 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 [135].

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 [32, 108, 109].

3.3 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 [104, 134, 138] this section summarizes the most relevant features.

3.3.1 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 3.1, 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.

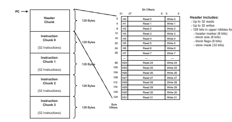

Figure 3.1: TRIPS Block Format.

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.

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 con-

straints [137].

3.3.2 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 3.2, 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 3.1.

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. Nagara-

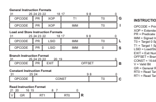

Figure 3.2: TRIPS Instruction Formats.

jan 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 of dynamic) instructions are issued [109]. Burger et al. [29] classify other architectures according to this terminology.

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 in which loads and stores must execute. The TRIPS ISA manual contains a complete description of the instruction set architecture [104].

3.4 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 a scalable and distributed is a processor that has no global wires, is built from a small set of reused components stitching me mated networks, and can be extended to a wide-issue implementation without resorting to centralized structures or redesigning the ISA.

The three dynamic execution 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 allowed a hierarchical design flow at all stages of the design – design specification through RTL coding, verification, and physical design. Modularity refers simply to the logical uniformity of the architecture through a small set of units; tiling refers to a physically regular placement and interconnection among these modules.

Interconnection networks: The tiles [modules] communicate through well-defined interconnection networks, which in turn have well defined flow

control properties and prevent deadlock avoidance and scalability properties [84].

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

3.5 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 3.2 shows the tile-level block diagram of the TRIPS prototype. The three major components are the chip, one or more memory, and the secondary memory system. The processor core occupy the top and bottom-right quadrants of the chip, and the secondary 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 unified (data and instructions) 1-MB NUCA cache, two integrated SDRAM controllers, a DMA controller, two chip-to-chip (C2C) routers 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 secondary memory are connected internally by one or more micronetworks. We define micronetwork

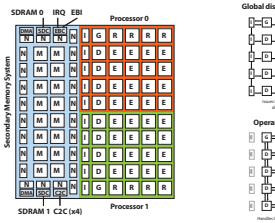

Figure 3.2: TRIPS Prototype Chip Schematic

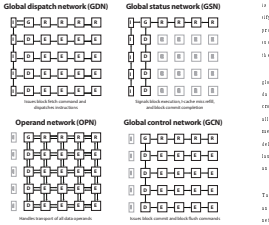

Figure 3.4: TRIPS Micronetworks (GRD, DSN, and ESN not shown).

us: "a network that employs many of the traditional networking techniques, such as flow control, but which implements a micro-architecture 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 protocol used [89]. Taylor et al. describe another taxonomy for classifying such micronetworks based on a topology's ability to deliver different types of messages from sources to destinations [86].

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 3.4. It connects all the tiles except for the ITs in a two-dimensional, wormhole-routed, 5x5 mesh topology. The OPN has a 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 operand packet to accelerate wakeup and select for typical operands that traverse the network.

Each processor core contains six other micronetworks as described in Table 3.1. Links in each of these networks connect only nearest neighbor tiles and messages traverse one tile per cycle. We show the link further if these networks in Figure 3.4 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-issue execution, 64KB of L1 instruction cache, 32KB of L1 data cache, and 4 SMT threads. The microarchitecture sup-

Micronetwork	Function
Operand network (OPN)	Point-to-point operand traffic between tiles
Global dispatch network (GDN)	Dispatch instructions to tiles
Global control network (GCN)	Commit and flush signals
Global status network (GSN)	Transmission of information about block completion
Global refill network (GRN)	I-cache miss refills
Data status network (DSN)	Communication of store completion status among the L1 data cache tiles
External store network (ESN)	Determine the completion status of stores in the L2 cache or memory.

Table 3.1: TRIPS processor micronetworks.

ports up to eight TRIPS blocks in flight simultaneously, seven of them speculative. If a single thread is running, seven 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 network, with 16-byte data links and four virtual channels. The micronetwork is optimized for cache-line sized transfers: four header packets followed by four 16-byte data packets), although other request sizes are supported for operations like loads and stores that do not require a cache line. The OCN acts as the transport fabric for all interprocessor, L2 cache, DMA, I/O, and DMA traffic.

In the rest of this section, we describe the contents of each processor

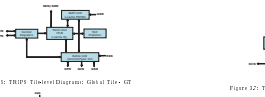

Figure 3.5: TRIPS Tile-level Diagrams: Global Tile - GT

Figure 3.6: TRIPS Tile-level Diagrams: Register Tile - RT

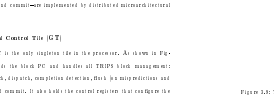

Figure 3.7: TRIPS Tile-level Diagrams: Data Tile - DT

Figure 3.8: TRIPS Tile-level Diagrams: Execution Tile - ET

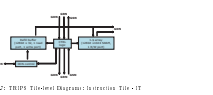

Figure 3.9: TRIPS Tile-level Diagrams: Instruction Tile - IT

processor in different operations, executions, and threading modes. Then, the GT interacts with all of the control networks, as well as the OPN for routing and writing the block PC. The major functions in the GT are the instruction cache tag array, the instruction TLB, and the next-block predictor [121].

3.5.1 Instruction Tile (IT)

The IT is simply an 8-kB slice of cache banks for the GT, which holds their tags. As shown in Figure 3.2, each IT contains a 2-way, 8KB bank of the L1 I-cache. Since each TRIPS block can consume as many as 128 bytes* worth of instructions, the microarchitecture treats blocks into five 32-instruction chunks, making each chunk is one cache line. Each 8KB IT bank can thus hold a 128-byte chunk from each of 128 blocks.

3.5.2 Register Tile (RT)

Centralized register files cause power and delay problems in large, wide-issue 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.

Since many 4-wide pairs of instructions are constructed in its read/write [Ds]) 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 forwarded by the memory system directly to the address and data fields of the store. If the load predicate is true, however, the null instruction instead fires, also targeting the address and data fields of the same store word. Note that although two instructions are targeting each operand of the store, only one of these instructions will fire due to the predicate. When there is a load in the processor DT and there is an outstanding null 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 arrived. Nullified register writes and stores are used to ensure that the block always produces the same number of outputs for completion detection.

3.7 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 hierarchy 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

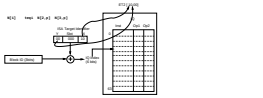

Figure 3.10: TRIPS execution example.


```
R[0]    read R4          N[1,L] N[2,L]
N[1]    mov1 #0          N[3]
N[2]    teq              N[2,p] N[3,p]
N[2] p_f  muli #4        N[32,L]
N[2] 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
```


3.5.3 Data Tile (DT)

The four DTs, each of which is a client on the OPN, each hold one 2-set, 8KB bank of the 32KB L1 data cache, as shown in Figure 3.2. Virtual addresses are interleaved across the DTs like at the granularity of the D-cache 64B cache-lines. In addition to the L1 cache bank, each DT contains a copy of the load/store queue (LSQ), a dependence predictor, a non-unity hardwired round-robin write buffer, a data TLB, and a MSHR that can support up to 16 requests for a 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 is much DT uses a 1024-entry bit vector. When an aggressively issued load causes a dependence misprediction (and subsequent pipeline flush), the dependence prediction 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 vectors entries in this scheme, the load wave clears the dependence predictor after every 16,384 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 most common problems are (a) determining how to distribute the LSQ among the DTs, and (b) determining when all earlier stores have completed—since all DTs—s that a load could issue.

We solved the LSQ distribution problem largely by brute force. Corralling the LSQ would have resulted in poor performance and too much complexity, so load would have to be issued to two places and the system resizes as the appropriate entries. Partitioning the LSQ among the DTs was problematic since we had no low-overhead solution for handling overflow of any of the partitions. Instead, we replicated four copies of a 256-entry LSQ, one at each DT. This solution is unreliable and wasteful since the maximum occupancy of all LSQs is 25%), but was the least complex alternative for this prototype. The LSQ can accept one load or store per cycle, and forwards data from earlier stores as necessary. If there is a partial (4-flight match (e.g., multiple stores byte instructions flowing a single, latest load word instruction), the load consumes one cycle for each store that forwards a piece of the load.

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

3.5.4 Secondary Memory System

The TRIPS prototype supports a DDR-266 static NUCA [66] memory, organized into 16 Memory Tiles (MT), 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

Event	Microarchitecture	Tiles	Description
Fetch	OPN	GT, IT	Fetch if it is cache hit in cache, if not send commands to ITs to fetch block from external memory system into its cache
Dispatch	GDN	GT, IT, RT, DT, ET	Send instructions from instruction cache's banks to different tiles
Execute	OPN	ET, RT, DT	Instructions execute in data flow fashion (ET, DT)
	DSN	DT	DTs' use its DSN network for maintenance dependences
Completion or Exception			
Completion	GSN	RT, DT, GT	ET sends a completion command to RTs' and DTs; RTs and DTs send a complete command to the GT when all register writes are received at the RT and DTs respectively
Exception	GSN	RT, DT, GT	If exception detected on a memory access or read, instructions is passed on the GT
Commit or Flush			
Commit	GCN	RT, DT, GT	GT sends commit command to RTs' and DTs; receive acknowledgement when architecture state completely updated. This two-phase commit, commit-acknowledge ensures the logical view of atomic block commit
Flush	GCN	RT, DT, GT	RTs' and DTs' send acknowledge command when architecture state completely updated. This two-phase commit, commit-acknowledge ensures the logical view of atomic block commit
Commit-ack	GSN	RT, DT, GT	RTs' and DTs' send acknowledge command when architecture state completely updated. This two-phase commit, commit-acknowledge ensures the logical view of atomic block commit

Table 3.2: Block execution timeline and microevents.

Figure 3.11: Example of a single instruction and mapping instructions to reservation stations.

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-flight block that writes to R4. That value is sent to the left operand of two instructions, the teq (N[1]) and the mov1 (N[1]).

When the test instruction receives the register value and the immediate '0' value from the mov instruction, it fires and produces a predicate which is routed to the predicate field of N[2]. Since N[2] is predicated on false, if the routed operand has a value of 0, the mov1 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, it will never execute in left operand.

If the load fires, it sends a request to the previous DT, which responds with the value of the load and routes it to N[33]. The DT uses the load/store

Figure 3.12: TRIPS execution example.

are the global reset signal, the "processor halted" signal from the GT, to the external bus controller (EBC), and the "processor halt" command from the EBC to the GT's. 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. If system-on-chip (SOC) designs that aggregate components with different functions via a portable communications network or bus, and (2) with chip-multiprocessor designs, in which a processor can be replicated many times on the chip, both are hierarchical and distributed designs. TRIPS differs from SOCs and CMPs in that the individual tiles are designed to have diverse functions but to operate together to implement a more powerful and design-scalable microprocessor. 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.

3.7.1 Chip Specifications

The TRIPS chip is implemented in the IBM CU-11 ASIC process, with baseline features size of 130nm and 7 layers of metal. The chip itself includes more than 170 million transistors in a chip area of 18.3mm 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 tools including test infrastructure insertion, the final physical floorplan, placing and routing of all wires, and the tapeout process.

The final clock period at worst-case process parameters is 4.2ns, which accounts for pessimistic clock skew and wiring parasitics from the final layout. To first order, this corresponds to approximately 32 fanout-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 8-12 FO4 [8]. A custom design style coupled with a more aggressive design team and more time devoted to timing optimization would likely enable to shrink the TRIPS implementation into that same region. Adding a more aggressive process and less conservative gates than a standard ASIC process would make TRIPS clock rate competitive with that of a high-end commercial microprocessor.

Figure 3.13 shows an annotated floorplan diagram of the TRIPS chip taken directly from the design database as well as a coarse area breakdown by function. The diagram shows the boundaries of the TRIPS tiles, as well as the placement of register and SRAM arrays within each tile. The left-most is 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 3.3 lists the area breakdowns of the major components of the chip.

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

Figure 3.13: Floorplan diagram

Overall Chip Area	
18%	Processor 0
18%	Processor 1
1.8%	Level-2 Cache
10%	On-chip Network
7%	Other (routing, etc.)
Processor Area	
11%	Instruction Cache (I-Cache)
4%	Register Files and Queues
20%	Level-1 Caches (I and D)
13%	Instruction Queues
13%	Load/Store Queues
12%	Operand Network
4%	Branch predictor
3%	Other

Table 3.3: Chip area breakdowns

network (OCN). Two two 133/266MHz DDR SDRAM controller (SDC) each connect to an individual 8GB SDRAM DIMM. The clock-multiply controller (CMC) extends the on-chip network to a four-port mesh router that globally connects to other TRIPS chips. Two clocks 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 in up to 32 TRIPS chips on 8 boards. Two two-channel 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 the 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 process system calls for the TRIPS chips on the board.

IOs and Test: The TRIPS chip includes nearly 600 signal I/Os, including 288 for each SDRAM interface, 210 for the chip-to-chip controller (10 pins per channel x four directions x input and output), and 40 pins for the EBC. Not shown in Figure 3.12 are the individual I/O cells, which are placed near the periphery of the chip. Some of ET-, MT-, and DT- 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 used our IBM partners added a custom feature to enable the core clocks to be used for ring clocking 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 three on-chip clock domains: main clocks, C2C clocks, and two clocks for the DDR SDRAM controllers. We assume that these clocks are asynchronous to one another and 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.

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 our specific architecture and as a research resource to implement and evaluate these mechanisms. Choosing a specific ISA and microarchitecture is necessary for quantitative evaluation. This ISA and microarchitecture are also inherently tuned to support polymorphism. The following graph abstraction is the TRIPS ISA directly leads itself to polymorphism at all corners as the unifying abstraction levels: express concurrency at different granularities. The distributed and modular nature of this microarchitecture already provides the coarse grain 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 process or microarchitecture, but the basic mechanisms could be applied to any architecture.

4.1 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

3.7.2 Physical Design

The TRIPS design flow relies heavily on tile-level partitioning as well as a modular ASIC design flow. As a part of their ASIC version, IBM provider engineer and SRAM array generators that were used heavily not only for memories and memory, but also for block prediction tables, instruction queues, and reservations stations. Through a university license, Synopsys provided their Design Ware suite which included synthesizable integer units, floating-point units, queues, and FIFOs. The design-time design part of is 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 3.4 shows additional details on the design of each TRIPS tile. The *Cell Instances* columns shows the number of synthesizable instances in each tile, which provides a relative measure of the complexity of the tile. *Array Bits* indicates the total number of bits found in dense register and SRAM arrays on a per-tile basis, while *Total bits* shows the area of a representative of each type of tile. *Total transistors* 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 3.4, the ET 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 instruction and data is viewed somewhat by the CAM array for its maximum simultaneous load/store

Tile	Function	Cell Instances	Array Bits	Max Array Bits/Tile	Total Tile Instances	% Chip Area
GT	Processor control	52,854	93k	1.2	1	3.6
IT	Instruction cache	5,455	135k	1.2	10	5.9
DT	L1 Data cache	119,912	89k	6.2	8	27.8
ET	Instruction execution	84,897	13k	2.3	32	28.0
RT	L2 Data cache	89,115	542k	6.2	16	30.2
NT	OCN NW interface and routing	23,687	-	1.0	24	7.2
SDC	DDR SDRAM controller	30,254	6k	5.2	2	3.4
DMA	DMA controller	28,846	4k	1.8	2	3.3
EBC	External bus controller	29,213	-	1.2	1	0.9
C2C	Chip-to-chip command/data network	37,211	-	2.2	1	4.2
	Clock tree and pads	1,552	1.5k	-	111	10.2

Table 3.4: TRIPS Tile Specifications.

queue which had to be implemented from discrete clusters, because no suitable dense array structure was available. We view the same phenomenon in OPN and OCN routers. The large cell count in the ET are due largely to the computational units, such as the floating point units, which are synthesized in the standard cell library rather than implemented using a custom datapath.

3.7.3 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 replicable test-bed bench for their tile that employed both directed and random tests to exercise as many of the corner cases as possible. Random tests ranted both test inputs and the timing of responses to the requests. To assess coverage, we augmented each tile design with event monitors, and

ensured that the counters were examined, 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 its full chip.

We also spent four person-months on performance validation. Using a suite of micro-benchmarks, with some randomly generated programs, we reduced the average error between the low-level performance simulator and the RTL simulator from 20% on average to 2%. This effort uncovered various 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 block penalty by one cycle, and reordering predictor operations to eliminate an occasional pipeline bubble before issuing a fetch.

3.8 Discussion

In this chapter we described the EDGE ISA, the TRIPS ISA which is one instance of an EDGE architecture, the 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 prototype chip provides limited polymorphism support, namely, explicit thread-level parallelism by sub-dividing the microarchitecture windows, reconfiguration of memory banks to provide programmer-controlled scratch-pad support, and DMA controllers that coordinate data movement between processors and memory. In the following chapter, we develop the principles of polymorphism and explain the mechanisms in the context of the TRIPS prototype architecture.

Chapter 4

Polymorphism in the TRIPS Architecture

Emerging applications with heterogeneous computation needs and diverse technology constraints have created a need for designs matched to their unique requirements. Ideally, a single processor could dynamically configure itself to exploit the appropriate types and granularities of parallelism in an application, and could provide the appropriate memory system resources. Such a processor would be able to adapt to the different types of parallelism and locality found in different applications, and could provide high performance and efficient execution for a wide range of applications. Unfortunately, designing a processor that can dynamically adapt to the wide range of behaviors found across different applications is extremely challenging. Instead, designers typically build specialized processors that are optimized for a narrow range of applications.

Processors can be specialized for different types of parallelism, such as instruction-level parallelism (ILP), thread-level parallelism (TLP), or data-level parallelism (DLP). Processors can also be specialized for different memory behaviors, such as supporting cache coherence for multiprocessors, supporting large cache capacities, or supporting streaming memory accesses. Specialization can also occur at the circuit level, where circuits are optimized for different clock rates, power budgets, or reliability requirements. Specialization can occur at the system level, where systems are designed with different numbers of processors, memory capacities, and I/O bandwidths. Specialization can occur at the software level, where software is written to target a specific programming model, such as shared memory or message passing. Specialization can occur at the application level, where applications are written to target a specific application domain, such as graphics or scientific computing.

Specialization can provide significant benefits in terms of performance, power, or area efficiency, but specialized processors can be inflexible when faced with applications that do not match the specialization model. Specialized processors can also be inefficient when executing applications that require a mix of different types of parallelism or memory behaviors. For example, a specialized processor that supports only TLP will be inefficient when executing an application with little or no TLP. Similarly, a specialized processor that supports only cache-coherent memory will be inefficient when executing an application that would perform better with a streaming memory system.

Many computer systems are specialized to efficiently execute a single application, or a single application domain. For example, cell phones 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 a peripheral/I/O controller. Several embedded manufacturers expect a multitude of processing tasks on a single device: wired Ethernet, wireless Wi-Fi, and cellular 3G communications, storage management, biometric identification, security and digital rights management, 3D sound field, and 3D video processing to name a few [38].

Designing multiple-profile solutions introduces a processor complexity problem. Architectural polymorphism solves this application heterogeneity problems and addresses technology constraints in a complexity-effective manner. We defined polymorphism in chapter 3 as "the ability to modify the functionality of coarse grain 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 Olukotun's definition of complexity-effective processor design [88]:

> A complexity-effective design is a design that: 1) embraces a multi-level model of scheduling design principles and associated mechanisms, and 2) has been validated by reducing unnecessary complexity into the areas the fundamental and elegant mechanisms.

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 completely 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 simple data-level parallelism and operates on two long arrays. One fixed function extension is to build a vector item-add interface in a conventional processor and compile programs into vector instructions. On the other hand, the polymorphous mechanisms in a more mechanistic for a conventional processor to modify the instruction fetch, order and execution logic to provide instruction efficiency and modify the memory system to provide support for regular memory accesses. 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 configure the microarchitecture differently to support different granularities of parallelism. The following two sections describe the resources and the mechanisms.

Granularity of configurations: A polymorphous architecture can allow behavior of coarse-grain microarchitecture modules, by changing the control logic between data-path and storage elements. Providing application specialization by configuring the grain blocks can be a challenge. Even efficient reconfigurable functionality from grain processing units is cryptanalysis blocks with different functionality to provide application-to-application specialization of hardware. They have all mostly provided application specific hardware and not programmable hardware. As mentioned in chapters 2 and 2, examples include FPGAs, Tensilica, Pact-XPP, MathStar, Piperench, and ARM AI of these designs would swell the small domain of problems where the application can be easily mapped to the hardware, typically 'regular' applications that perform mostly on general purpose programs. By integrating an FPGA to a conventional processor pipeline, the Garp architecture performs fine grain configuration on this hybrid programmable substrate [28]. 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. This chapter describes the mechanisms which create a configurable execution core, configurable control flow management, 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 each an approach provides, and in chapter 9 in the conclusion of this dissertation, we provide a broader discussion comparing polymorphism to other approaches.

4.2 Resources

We classify the type of resources in polymorphous architecture into three categories based on their function. In the next section we identify these resources and describe the mechanisms for configuring these resources.

Fixed resources: Some resources in the processor operate in the same way regardless of the application executing on the processor. 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 provide different types of operations or change their operating policies, depending on program behavior. For example, microarchitecture logic either fetches from one single program thread all the time, or uses 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 resources. In an SMT processor with support up to 4 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 specialization 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 classifications. 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 IBM Cell processor's SPEs can be considered specialized resources since they are primarily used to execute single precision SIMD code where data has already been brought into neighboring memory banks [30]. Today's multicore chips and the SUN niagara [9] can be viewed as homogeneous systems with only fixed resources providing a single execution model to all programs.

4.3 Mechanisms

The TRIPS ISA expresses concurrency to the hardware by breaking programs into blocks and encoding instruction dependences within these blocks by marking the data flow 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 register.

4.3.1 Execution core management

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

Across different granularities of parallelism, the structure of these dataflow graphs can vary, and the types of communication between these dataflow graphs can change as well. With respect to this, we where ILP is the dominant type of parallelism, the size of the graphs is quite small - of the order of 20 to 40 instructions. When executing multiple programs, dataflow graphs from different programs must be managed in the execution core for correct TLP, and when there is ample data-level parallelism, these graphs can be very large. To extract ILP efficiently, the reservations stations are used to map a small number of specifically fetched dataflow graphs, since these graphs are typically small and many such graphs are needed to fill the reservation stations space. To extract TLP, the reservation stations are partitioned across programs and dataflow graphs from multiple programs are mapped in the reservation stations. 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.

4.3.2 Control flow management

Depending on the type of parallelism, the control behavior of applications may quite dramatically. These control flow mechanisms may be 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. With 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 in execution. Instead, we design an optimized instruction fetch mechanism that senses instructions that have been fetched once for a block regularly.

These control flow techniques are not mutually exclusive. By using limited amount of control speculation within each thread while extracting TLP, the processor efficiency can be further increased. Some programs with

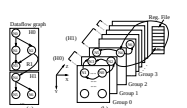

Figure 4.2: Execution core management for ILP.

this is achieved by specializing the instruction fetch between blocks. The basic mechanisms 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 processors. Second, we map the reservation stations in the execution core like a circular buffer and map multiple blocks in the instruction window and execute instructions across these blocks simultaneously. The next-block predictor is a specialized resource and the reservation stations form a polymorphous resources, both of whose functions are described below.

Next-block prediction The next-block prediction is made using a multi-hop tournament exit predictor [8], which predicts a binary value indicating the branch that is predicted to be the exit of the block - small branch block

outputs.

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 of the fetch block. 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 larger target field, allowing hard to predict branches to be predicated, thus hiding control flow misprediction penalties. The two main challenges in exporting 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 non-nop 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 allows us to push the compiler to its limits and explore the design space.

4.4.2 Control flow management

To enable an effective large instruction window in 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 have between 20 and 60 instructions on average. However, to extract ILP, a much larger window of instructions must be examined and

4.4 Instruction-Level Parallelism

In this section, we describe how polymorphism can be used to run single-thread code 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 [32].

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 input encoding in the TRIPS ISA enables out-of-order execution while avoiding the associative issue window logic of conventional machines. To use the instruction bandwidth efficiently 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.

4.4.1 Execution core management

The polymorphous resources in the execution core are the reservation stations that provide instructions and operand storage space. To extract ILP, these reservations stations are configured to issue instructions in instruction windows. Such a configuration uses the reservations stations at each Execution Tile to map dataflow graphs directly in the ETs. This physically distributed issue window spread across the ET allows orders of magnitude increases in window sizes compared to conventional superscalar processor designs - as the TRIPS implementation we achieve an order of magnitude increase. Since there are multiple reservation stations at each ET and multiple ETs, this window is fundamentally a two-dimensional window (as the grid-like connections correspond to the physical dimensions of the ET array and the two dimensions corresponds to multiple instructions slots at each ET, as shown in Figure 4.3.

To fill one of these relatively large instruction windows, the compiler breaks its 3-D region, assigning each instructions in 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 a break block instruction would begin in group of 128, with each 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 4.4a shows a four-instruction block (BB) mapped into the first group of reservation stations. Figure 4.4b 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 operands are 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

Parallelism	Resources	Policies
Execution core management		
ILP	Reservation stations	Map multiple dataflow graphs (1 at time)
TLP	Reservation stations	Map multiple dataflow graphs from different threads
TLP	Instruction fetch/next-block logic	Prioritize between threads
DLP	Reservation stations	Map large unrolled dataflow graph (1 at time)
Data storage management		
ILP	Load/store ID bits	Register renaming across blocks
TLP	Register files	Storage for architecture state from many threads
DLP	Register files	Hold register file base words
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 - store fetched instruction

Table 4.2: Summary of polymorphous mechanisms.

4.4.3 Data storage management

Memory accesses in DLP programs are dominated by regular patterns, typically load or stored strides. However, significant number of other types of data accesses are also present, including irregular accesses in small lookup tables, accesses to a large number of read-only 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 accesses.

Short-term data: The management of short-term data is identical to what is done in current ILP. The large size of graphs typical when programs have DLP does not make any difference in the way most of these operands are managed. The fixed-size register memory access times by these programs present an opportunity for optimizing some short-term data accesses. When performing strided memory accesses, individual load and store instructions that implement this strided access in the dataflow graph, view regularity as well. Vector instruction sets include some form of a load instruction that can read multiple words of data from memory and write in a vector register file. Similarly, a multi-word load instruction can be used to fetch multiple words from memory and send them packed to reservation stations in the ETs. This instruction efficiently encodes strided access and amortizes the per-memory instruction overheads which include the execution overheads of multiple load instructions, the communication overheads of routing multiple address to the

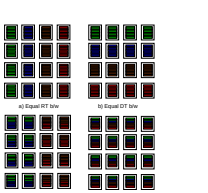

Figure 4.3: Partitioning execution resources to support thread-level parallelism. Each color denotes a different thread.

The partitioning of these resources raises two questions:

- When to partition: Static partitioning is straightforward 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. Empirical investigations and policies to the hardware introduces software complexity and dynamic partition of processor resources introduces hardware complexity. Both ways 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 4.3 shows a spectrum of partitioning strategies. The main differences between the partitioning schemes are implementation complexity, thread distance from the register files across threads, thread distance from the data cache across threads, thread interaction for load/store and interacy. The partitioning strategies shown in (a), (b), and (c) in figure 4.3, add complexity to the instruction fetch logic as the natural alignment of 32 instructions per bank must be changed, as the instruction fetch network must be augmented to route instructions across rows. Figure 4.3d shows the strategy adopted in TRIPS which leaves most of the design unchanged and requires modifications only in the instruction selection

logic in the core. Since the TRIPS ISA has fixed 128-instruction blocks, any sliced 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 its ILP model as possible, and to reduce implementation complexity, in the TRIPS prototype chip we implemented a simple sharing scheme denoted in Figure 4.3d. Each thread gets 1/4th of the resources and up to 4 threads can be executing simultaneously.

4.5.2 Control flow management

Control flow management mechanisms to support thread-level parallelism is not very different from the mechanisms used for ILP, namely control flow speculation and high bandwidth instruction fetch. The added requirement in that hardware 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. Since the exit history is crucial to high accuracy prediction, we implement the global history GHR registers and maintain one copy for each thread. The return-address stack (RAS) is also replicated for each thread. The return instructions in different threads 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 48 bits of storage (80 bits per thread), the resulting replicated storage is quite small.

High-bandwidth instruction fetch: The management of the instruction caches and the network in streams 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 its state as which threads complete. Tullsen et al. investigate several policies that can implemented for instruction fetch between multiple executing threads [34]. 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.

4.5.3 Data store management

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

Long-term data: To support multiple threads executing on the same processor core, enough replicated register storage must be provided to maintain the architecture state of each executing thread. One copy of the architecture

register is provided for each thread. One could argue that replicating register file storage is expensive and not in the spirit of polymorphism if that storage cannot be used for anything else. The register renaming hardware must be aware that values should not be forwarded across threads, which is achieved by changes in only the control logic of the register renaming hardware. No replication of temporary storage or datapath is required in generic thus necessitating a register file.

Persistent-data: The memory system operates much the same as when executing ILP. Similar to modifications in 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.

4.6 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 memory patterns, poor locality but tolerance to memory latency, and high computation intensity [36]. The dataflow graph abstraction already lends itself 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 principle, programs with DLP can be executed on the TRIPS processor by ordering an control flow speculation and having the hardware extract ILP. In chapter 7 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 an ILP-like execution environment under the following three categories: the execution core, control flow, and data storage.

4.6.1 Execution core management

For programs with ILP and TLP, the dataflow graphs were typically small and control-flow prediction was employed multithreading to memory 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 execute DLP is to unroll the graphs as much as possible and map large unrolled dataflow graphs to the reservation stations, without relying on speculation.

4.6.2 Control flow management

Control flow efficiency 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 removing 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 Single Instruction Multiple Data (SIMD) execution, where the dataflow graph can be viewed as one single SIMD instruction executed across multiple ALU cores. The overheads of repetitive instruction fetch and unnecessary operations 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 revive mapped instructions and augment the fetch logic to fetch instructions in a loop just once.

Also, with some types of DLP programs, a fine grain multithreaded model that provides a Multiple Instruction Multiple Data (MIMD) execution model is preferred. The ILP and TLP execution model of exposing a program counter that fetches and maps successive dataflow graphs (sometimes through control speculation) is not very efficient compared to this approach because models. By adding instruction storage support and augmenting the ALUs independently the execution core can be tailored to look like a MIMD array and achieve its instruction fetch efficiencies.

4.6.3 Data storage management

Memory accesses in DLP programs are dominated by regular patterns, typically load or stored strides. However, significant number of other types of data accesses are also present, including irregular accesses in small lookup tables, accesses to a large number of read-only 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 accesses.

Short-term data: The management of short-term data is identical to what is done in current ILP. The large size of graphs typical when programs have DLP does not make any difference in the way most of these operands are managed. The fixed-size register memory access times by these programs present an opportunity for optimizing some short-term data accesses. When performing strided memory accesses, individual load and store instructions that implement this strided access in the dataflow graph, view regularity as well. Vector instruction sets include some form of a load instruction that can read multiple words of data from memory and write in a vector register file. Similarly, a multi-word load instruction can be used to fetch multiple words from memory and send them packed to reservation stations in the ETs. This instruction efficiently encodes strided access and amortizes the per-memory instruction overheads which include the execution overheads of multiple load instructions, the communication overheads of routing multiple address to the

cache, and the memory access overhead of sending each word from the cache.

Long-term data: Accessing register values can become a bottleneck, if one register value has a lot 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 instance is inefficient 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 spatial revitalization where systematic fetch and charging during multiple iterations of a dataflow graph are read once and stored multiple times, instead of being repeated read from the register file thrashing the contents of register read and remains. This mechanisms is not restricted to DLP, and can be utilized while executing ILP or TLP if the compiler can statically determine this behavior.

Persistent data: To support DLP, a software managed cache memory building the on-chip memory like a better than hardware managed instruction caching. Other designs like Smart Memories, Imagine, and the IBM Cell processor have adopted this approach. To behave as a software managed memory, the reconfigurations of the memory tiles include 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 to enable block transfers between the tile and remote storage (such as memory or other tiles), virtual access to remote storage (gather/scatter), and indirect gather/scatter in which the source and destination addresses are contained within a cache of the data's storage. Instead of using the processor to orchestrate these transfers, a core-level DMA controller is integrated on chip to perform these functions.

4.7 Discussion

In this section we have described the principles of polymorphism and a case-set of fundamental mechanisms to support instruction-level parallelism, thread-level parallelism, 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 a level effect 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 build speculative instructions, with a next-block predictor is specialized resource used to perform control flow prediction. For TLP, which is coarse grain 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 overhead of ILP style execution in this

chapter. Chapter 7 includes a detailed analysis of DLP program behavior and the synthesizes of polymorphous mechanisms for DLP.

Polymorphism comes as a natural way to address processor complexity and technology constraints and addresses 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 5

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 codes with different kinds of complexity and different types of behavior to quantitatively evaluate the TRIPS design and demonstrate its effectiveness. We start with a set of low-level or mechanical-math 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 mechanisms. We then employ a set of data parallel kernels and the EEMBC simulated 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, where programs are significantly more complex than the EEMBC benchmarks.

In Section 5.1 we describe the methodology of this ILP study and tools used in our performance evaluations. Section 5.2 describes the benchmarks. Section 5.3 discusses the performance results.

5.1 Methodology

To evaluate the performance of the TRIPS processor in advance of the silicon, we developed a detailed cycle-level simulator, called tsim-proc, which models the hardware at a much more detailed level than high-level simulators like SimpleScalar [26]. 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 (tsunami) [20] to attribute percentages of the critical path of the programs to different microarchitectural activities using the technique first proposed by Fields et al. [18]. 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 5.1 lists 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

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 5.1: TRIPS processor parameters

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 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 [40].

5.2 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

List of Benchmarks	
Microbenchmarks: sha, dct8x8, matrix, vadd	
Data Parallel Benchmark Kernels	
Scientific Computing	LU, FFT
DSP	convert, fft, 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 175.vpr 181.mcf 197.parser 256.bzip2 300.twolf	168.wupwise 177.mesa 179.art 200.sixtrack 301.apsi

Table 5.2: Benchmarks

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 micorbenchmarks, 2) a set of data parallel benchmarks(DLP) kernels we developed, 3) the EEMBC suite [44], and 4) the SPEC CPU2000 suite [144]. Table 5.2 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 computed 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 7 and we include 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.

All these benchmarks were compiled using the TRIPS compiler toolchain which takes C or FORTRAN77 code and produces complete TRIPS binaries

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 5.3: TRIPS performance results on microbenchmarks.

that will run on the hardware. Although the TRIPS compiler is able to compile major benchmark suites correctly (i.e., EEMBC and SPEC2000) [137], there are many TRIPS-specific optimizations that are currently being developed and incorporated into the compiler. Prior to completion of those optimizations, the TRIPS compiler performance will be inadequate because many of the TRIPS blocks are too small.

5.3 Results

5.3.1 Microbenchmarks

Table 5.3 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 show 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 clock 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 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 so 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**

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	0.4	3	0.01
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	0.6	5	0.00
scientific/fft	1.36	17	22	0.6	5	0.33

Table 5.4: Processor performance on DLP Kernels

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.

5.3.2 Data parallel kernels

Table 5.4 shows the performance obtained on the data parallel benchmark suite. These applications have ample DLP and are typically coded in specialized 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 5.4 the processor is able to extract significant amount of ILP - the IPCs range from 0.29 to 4.92. 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 6 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 7 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 *fft* and *LU* 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 hand optimized version was 40% better than compiler generated code in that case.

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 7 we describe our experiments that compare the performance of specialized data parallel architectures to polymorphous DLP mechanisms.

5.3.3 EEMBC and SPEC CPU2000 benchmarks

Tables 5.5 and 5.6 show the performance obtained on the EEMBC and SPEC CPU2000 benchmarks. Most of the EEMBC benchmarks are very

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 5.5: 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 5.6: Processor performance on SPEC CPU2000 benchmarks

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 3.24. 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 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 inling, 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.

5.4 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 full 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 on line.

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 [107]. 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 focussed 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 6

Performance Evaluation: TLP

In this chapter we evaluate the performance of polymorphous mechanisms implemented in the TRIPS architecture for TLP. 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:** Partitioning of the reservation stations in the execution core between multiple threads. The TRIPS prototype chip implements a static partitioning approach in which each threads 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.

6.1 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 what we used for our 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.

6.1.1 Configurations

We study three processor configurations which are listed in Table 6.1. In all configurations $1/4th$ of next-block predictors storage tables are provided to each program with separate 12-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 unutilized, 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.

6.1.2 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

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 6.1: Different processor modes simulated

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 [133]. 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 generating 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 [154].

We exclude the four microbenchmarks from this study, as they are pri-

marily 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 programs 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.

6.1.3 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.

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. While the *ideal* configuration is the limit performance possible, the *max* configuration is maximum performance that can realistically be achieved given the physical resource constraints of the TLP mode. 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.

While discussing the speedup and efficiency of 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[1].

6.2 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 6.1 through 6.3 show results for the SPEC CPU2000 suite, Figures 6.4 through 6.6 show results for the EEMBC suite, and Figures 6.7 through 6.9 show results for the data-parallel benchmarks. Tables 6.2 through 6.7 shows the program mixes that were executed.

6.2.1 SPEC CPU2000 benchmarks

Utilization: Figure 6.1 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 pro-

[1] 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.

grams are executed serially in the processor in the ILP mode.

For the 2-Thread configuration, on average the IPCs are the same between the TLP-mode and ILP-mode. 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 is: 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.

1. **ILP-mode much better than TLP-mode:** In 14 of 40 mixes up to mix 12, the ILP mode of execution provides better processor utilization than the TLP-mode. With only two threads executing, only half the processor's reservation stations are used because of the simple partitioning strategy. Further each thread gets to execute one speculative block and one non-speculative block only. Specifically 4 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 they execute in such a configuration with an effective instruction window of only 1024. The 13 mixes corresponding to this case are dominated by these 4 benchmarks.
2. **ILP-mode slightly better than TLP-mode:** Another 7 mixes, from 14 through 20 perform slightly better in the ILP-mode. These are mixes 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.

3. **TLP-mode slightly better than ILP-mode:** Mixes 21 through 30 perform slightly better in TLP-mode. These are mixes where one application's performance is severely limited by the reduced instruction window, whereas one's is not.
4. **TLP-mode much better than ILP-mode:** 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 2 threads, and hence two source 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. The average IPC is approximately 3 and ranges from 0.27 to 3.44. Furthermore, compared to the ILP-mode, the TLP-mode is between 14% to 233% better, on average being 103% better. Clearly the processor is able to overcome the contention effects of sharing resources between multiple threads quite effectively. With four

available threads the processor has a large amount of useful work, at least 4 useful blocks every cycle. The workload mix in which the TLP mode does worse comprises emph 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. In the TLP-mode, the contention effects overcome the benefits of having more useful non-speculative work.

To summarize, the polymorphous mechanisms are able to effectively utilize the processor when executing 4 threads. When executing 2 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.

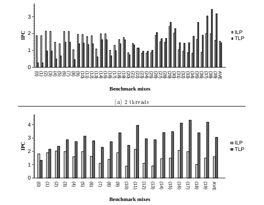

Figure 6.1: TLP-mode performance (utilization) - SPEC CPU2000 suite.

Speedup: Figure 6.2 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 6.1. 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 TLP mode does not fully utilize the processor because, only 2 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. 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 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 program serially. Figure 6.3 shows speedup achieved during executing in TLP mode, compared to concurrent execution of the multi-programmed workloads in ILP mode on multiple processors. Recall that, while the *ideal* configuration is the limit performance possible, the *max* configuration is maximum performance that can realistically be achieved given the physical resource constraints of the TLP mode. While executing 2 threads, on average an efficiency of 84% is achieved compared to the *max* configuration, and an average of 49% is achieved compared to the *ideal* configuration. While executing 4 threads, the efficiency surprisingly increases. The reason being that, the overheads of contention between the threads for resources is overcome, by the reduced effects of incorrect control speculation.

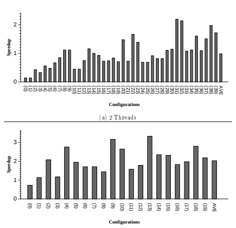

Figure 6.2: TLP-mode speedup compared to serialized execution - SPEC CPU2000 suite.

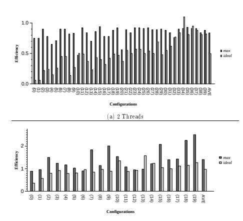

Figure 6.3: TLP-mode execution efficiency - SPEC CPU2000 suite.

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

Table 6.3: 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.

6.2.2 EEMBC benchmarks

The EEMBC benchmarks results show a trend very similar to the results for the SPEC CPU2000 benchmarks. We briefly summarize the results and our observations below. Figure 6.4 and shows the IPC comparison between ILP and TLP mode while running 2 threads and 4 threads. Fewer of the benchmarks have lots of parallelism, so more program mixes show a benefit from multithreaded execution (mixes 9 through 39 in 2-Thread configuration, and 1 through 19 in 4-Thread mode).

In terms of speedup and efficiency also the results are similar to the SPEC CPU2000 suite. Figures 6.5 and 6.6 show the speedup and efficiency data for the 2-Thread and 4-Thread configurations. The average speedup is 27%, and 86% for the 2-Thread and 4-Thread configurations, respectively. Both the 2-Thread and 4-Thread configurations are surprisingly efficient, sometimes, over-performing the ideal and max configurations. The reasons they are able to exceed the performance of stand-alone execution is that, because of reduced speculation depth, each thread executes less speculative work that gets discarded. The efficiency in the 2-Threads configuration compared to the *ideal* configuration is 140%, and 97% while compared to the *max* configuration.

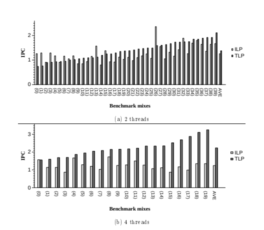

Figure 6.4: TLP-mode performance (utilization) - EEMBC suite.

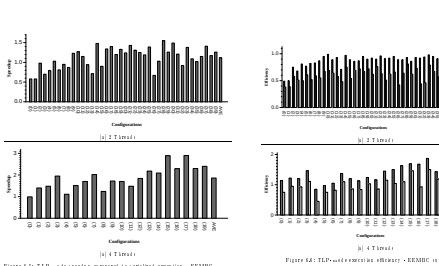

Figure 6.3: TLP-and-DLP speedup compared to serialized execution - EEMBC suite.

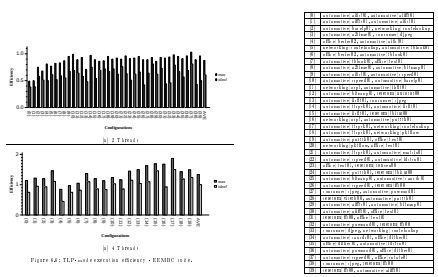

Figure 6.4: TLP-and-execution efficiency - EEMBC suite.

0	automotive-a2time01 + automotive-a2time01
1	automotive-a2time01 + automotive-bitmnp01
2	automotive-bitmnp01 + automotive-a2time01
3	automotive-bitmnp01 + automotive-bitmnp01
4	office-bezier02_automotive-a2time01
5	automotive-a2time01 + automotive-rspeed01
6	automotive-bitmnp01 + automotive-tblook01
7	automotive-rspeed01 + automotive-rspeed01
8	automotive-a2time01 + automotive-canrdr01
9	automotive-bitmnp01 + automotive-canrdr01
10	automotive-canrdr01 + automotive-canrdr01
11	automotive-a2time01 + automotive-ttsprk01
12	automotive-bitmnp01 + automotive-ttsprk01
13	automotive-canrdr01 + automotive-ttsprk01
14	automotive-ttsprk01 + automotive-ttsprk01
15	automotive-a2time01 + automotive-tblook01
16	automotive-bitmnp01 + automotive-tblook01
17	automotive-canrdr01 + automotive-tblook01
18	automotive-tblook01 + automotive-tblook01
19	automotive-a2time01 + office-rotate01
20	automotive-bitmnp01 + office-rotate01
21	automotive-canrdr01 + office-rotate01
22	automotive-tblook01 + office-rotate01
23	office-rotate01 + office-rotate01
24	automotive-a2time01 + automotive-pntrch01
25	automotive-bitmnp01 + automotive-pntrch01
26	automotive-canrdr01 + automotive-pntrch01
27	automotive-tblook01 + automotive-pntrch01
28	office-rotate01 + automotive-pntrch01
29	automotive-pntrch01 + automotive-pntrch01
30	automotive-a2time01 + automotive-aifftr01
31	automotive-bitmnp01 + automotive-aifftr01
32	automotive-canrdr01 + automotive-aifftr01
33	automotive-tblook01 + automotive-aifftr01
34	office-rotate01 + automotive-aifftr01
35	automotive-pntrch01 + automotive-aifftr01
36	automotive-aifftr01 + automotive-aifftr01

Table 6.5: Benchmark mix in 2-Thread configurations - 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 + automotive-a2time01 + automotive-a2time01 + automotive-a2time01
1	automotive-a2time01 + automotive-a2time01 + automotive-a2time01 + automotive-bitmnp01
2	automotive-a2time01 + automotive-a2time01 + automotive-bitmnp01 + automotive-canrdr01
3	automotive-a2time01 + automotive-a2time01 + automotive-bitmnp01 + automotive-tblook01
4	automotive-a2time01 + automotive-a2time01 + automotive-canrdr01 + automotive-canrdr01
5	automotive-a2time01 + automotive-a2time01 + automotive-canrdr01 + automotive-tblook01
6	automotive-a2time01 + automotive-a2time01 + automotive-tblook01 + automotive-tblook01
7	automotive-a2time01 + automotive-bitmnp01 + automotive-bitmnp01 + automotive-bitmnp01
8	automotive-a2time01 + automotive-bitmnp01 + automotive-bitmnp01 + automotive-canrdr01
9	automotive-a2time01 + automotive-bitmnp01 + automotive-bitmnp01 + automotive-tblook01
10	automotive-a2time01 + automotive-bitmnp01 + automotive-canrdr01 + automotive-canrdr01
11	automotive-a2time01 + automotive-bitmnp01 + automotive-canrdr01 + automotive-tblook01
12	automotive-a2time01 + automotive-bitmnp01 + automotive-tblook01 + automotive-tblook01
13	automotive-a2time01 + automotive-canrdr01 + automotive-canrdr01 + automotive-tblook01
14	automotive-a2time01 + automotive-canrdr01 + automotive-tblook01 + automotive-tblook01
15	automotive-a2time01 + automotive-tblook01 + automotive-tblook01 + automotive-tblook01
16	automotive-bitmnp01 + automotive-canrdr01 + automotive-canrdr01 + automotive-tblook01
17	automotive-bitmnp01 + automotive-canrdr01 + automotive-tblook01 + automotive-tblook01
18	automotive-bitmnp01 + automotive-tblook01 + automotive-tblook01 + automotive-tblook01
19	automotive-canrdr01 + automotive-tblook01 + automotive-tblook01 + automotive-tblook01
20	automotive-a2time01 + automotive-a2time01 + office-rotate01 + office-rotate01
21	automotive-a2time01 + automotive-bitmnp01 + office-rotate01 + office-rotate01
22	automotive-a2time01 + automotive-canrdr01 + office-rotate01 + office-rotate01
23	automotive-a2time01 + automotive-tblook01 + office-rotate01 + office-rotate01
24	automotive-bitmnp01 + automotive-bitmnp01 + office-rotate01 + office-rotate01
25	automotive-bitmnp01 + automotive-canrdr01 + office-rotate01 + office-rotate01
26	automotive-bitmnp01 + automotive-tblook01 + office-rotate01 + office-rotate01
27	automotive-canrdr01 + automotive-canrdr01 + office-rotate01 + office-rotate01
28	automotive-canrdr01 + automotive-tblook01 + office-rotate01 + office-rotate01
29	automotive-tblook01 + automotive-tblook01 + office-rotate01 + office-rotate01

Table 6.6: 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.

6.2.3 Data parallel benchmarks

Overall the data parallel benchmark kernels benefit very little from running in TLP mode. Figure 6.2 shows the IPC comparisons for the data parallel benchmarks. Overall only 4 of 46 program mixes perform better while running in the 2-Thread configuration and by only 8% between average, and 30 of 46 program mixes perform better while running in the 4-Thread TLP mode, and only by 25% between average. Figure 6.3 shows the speedup data for all the program mixes. 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, system network, and register file.

For these programs the efficiency of the TLP models also lose compared to the previous two suites. As shown in Figure 6.3, in the 2-Thread configuration, the average efficiency compared to the ideal configuration is only 4.3%, and 0.74% compared to the max configuration. The results for the 4-Thread configurations are similar, the average efficiency compared to the ideal configuration is only 42%, and a 7.4% compared to the max configuration. While one would expect the efficiency of 4-Thread configurations to be worse than the 2-Thread configurations because of more contention, that really is not the case. Between the 2-Thread and 4-Thread configurations, the amount of resources in both the TLP mode, and the ideal/max baseline increase linearly, while it only the efficiency remains relatively unchanged. The 2-Thread TLP mode does only half the processor 512 of the 1024 instruction windows slots is empty.

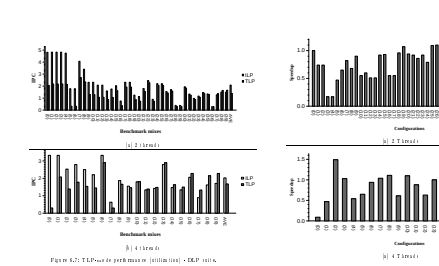

Figure 6.7: TLP-and-performance (utilization) - DLP suite.

Figure 6.8: TLP-and-speedup compared to serialized execution - DLP suite.

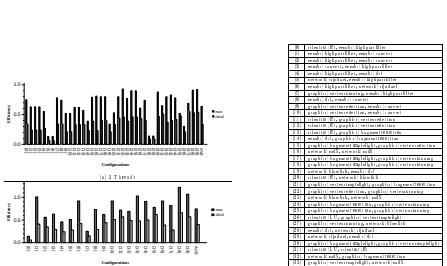

Figure 6.9: TLP-and-execution efficiency - DLP suite.

0	rotate01 + rotate01
1	automotive-tblook01 + rotate01
2	automotive-canrdr01 + rotate01
3	graphics-texturemapping-ransac01 + automotive-tblook01
4	automotive-tblook01 + rotate01
5	automotive-bitmnp01 + graphics-texturemapping-ransac01
6	graphics-texturemapping-ransac01 + rotate01
7	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
8	graphics-texturemapping-ransac01 + automotive-tblook01
9	graphics-texturemapping-ransac01 + automotive-canrdr01
10	graphics-texturemapping-ransac01 + automotive-bitmnp01
11	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
12	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
13	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
14	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
15	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
16	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
17	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
18	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
19	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
20	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
21	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
22	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
23	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
24	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
25	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
26	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
27	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
28	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
29	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
30	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
31	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
32	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
33	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
34	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01
35	graphics-texturemapping-ransac01 + graphics-texturemapping-ransac01

Table 6.7: 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.

Figure 6.10: TLP-and-execution efficiency - DLP suite.

6.3 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 bandwidth compared to the ILP-mode of the processor. By using multi-threading, the processor is able to effectively generate useful work and is often significantly better than using counters in particular to generate useful work from a single thread.

In fact, the reduced speculation depth helps tremendously in programs that have poor control prediction accuracy, coupled with small basic sizes and limited parallelism. By executing multiple threads, the processor resources are used to extract parallelism from different threads.

The primary hindrance in performance is that we expected was resource contention for the shared resources between the threads and 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 summarize the resource contention for the critical processor resources like the data cache ports, system network, and register file. As a result of contention at all these structures in the processor, the steering unit probes will eventually get stalled. Table 6.8 lists the percentage of cycles that the execution tiles are stalled due to resource conflict. We can see that while resource contention clearly increases, a significant difference is seen only for the SPEC CPU2000 benchmarks in the 4-Thread configuration.

Benchmark suite	ILP	2-Threads	4-Threads
SPEC CPU2000	24	28	58
EEMBC	28	35	37
Data parallel benchmarks	29	30	34

Table 6.8: Resource contention: percentage of cycles that the execution tiles are stalled due to resource conflict.

The results demonstrate that the polymorphous mechanisms are effective at executing 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 some architectural register file storage for the different threads and some 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 examine the resource contention for the critical processor resources like the data cache ports, system network, and register file. As a result of contention at all these structures in the processor, the steering pipeline will eventually get stalled. Table 6.8 lists the percentage of cycles that the execution tiles are stalled due to resource conflict. We can see that while resource contention clearly increases, a significant difference is seen only for the SPEC CPU2000 benchmarks in the 4-Thread configuration.

Chapter 7

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 single instruction like vector processors, systolic arrays, and SIMD arrays optimized for simple control and exploiting the regularity in the instruction streams and data streams. Such architectures had narrow application domains, but more recently to-tal 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 polymorphous execution. We first performed a detailed analysis of DLP workloads by characterizing their fundamental behavior in terms of memory behavior, control behavior, and computations. 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 7.1 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 7.2 we provide a detailed characterization of the fundamental behavior of DLP workloads and in Section 7.3 we evaluate DLP workloads using a conventional execution model to determine the bottlenecks that hinder DLP execution. In Section 7.4 we use the application characterization to develop a set of flexible microarchitecture mechanisms. Finally, in Section 7.5 we present performance results that can be obtained using these mechanisms and compare the results to specialized DLP architectures.

7.1 DLP Overview and History

Data-parallel programs are growing in importance, increasing in diversity, and demanding increased performance from hardware. Specialized hardware is common place in the mobile/embedded space, signal processing, and high-performance scientific computing domains. Modern graphics processors have rapidly evolved from 2d GPUs in 450 MHz in 2001 [25] to 3d GPUs in 650 MHz in the latest ATI Radeon R580, in late 2006. Based on these levels of performance we can expect that they have at least forty 32-bit floating point units. Software radios for 3G wireless baseband receivers are being developed for digital signal processors and require 15 Gops to deliver adequate performance [23]. Each arithmetic processor in the Earth Simulator contains forty eight vector pipelines and delivers peak performance of up to 8 GFlops. The IBM Cell processor in the Playstation3 system has a theoretical peak performance of 25.6 GFlops running at 3.2 GHz [9], and the Playstation 3 system has been reported as being able to provide 2 TFlops. The XboxS60 system has an estimated peak performance of 1 TFlop [8]. 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 complex data access methods to memory and memory is accessed more irregularly. SIMD architectures provide support for communication between execution units thereby enabling computations 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-grain processors and operate on larger chunks of data using the single-program, multiple data (SPMD) execution model, with poor support for fine-grain synchronizations. Emerging applications, such as real-time graphics, exhibit control behavior that requires fine grain MIMD execution and fine-grain 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 are organized

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 architecture two specialized vector units—one tuned for geometry processing in graphics rendering and the other specialized for vibrations and physical simulations [6], The Sony Handheld Engine requires 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 computations.

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 how 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 ranging from simple mechanisms, like efficient lookup tables, to different execution models, such as SIMD or MIMD.

Based on the programs attributes identified, we propose a set of polymorphous microarchitectural mechanisms for programming 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 evaluations. We also show enough comparisons of the performance of these mechanisms in current host-oriented 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 designs provides several relevant capabilities, including the ability to map various communication patterns and relatively placed dynamically issued execution, that enable a straightforward implementation of the mesh routines. 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.

7.2 Application Behavior

Data-parallel workloads can be classified into domains based on the type of data being processed. The nature of computations varies within a domain and across the different domains. The applications vary from simple computations on image data converting one color space to another (computing the luminance), to complex operations routines on network packets (computing SHA-1 of an entire file). Fixed functions never 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 as suggested 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.

7.2.1 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 converted to different execution units, operating on different parts of memory in parallel. We refer to this loop body as a kernel. To specify the iterations of the loop and to express data element reference and memory reference concurrency. 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. As a primarily varying number of transformation matrices are performed on each input vertex, a 3D mesh. 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) scanned 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 limited access or for example. We use the term record to refer to a group of elements on which a single invocation of a kernel operates. In image processing, for example, a record may consist of3 elements, corresponding to 3 primary color components. Because of the regularity of these accesses, microarchitecture can provide accesses or amortize the address calculations 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 manner, but the 2D DCT can be transformed into this model by unrolling all of the instructions for each 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 execution. Figure 7.1a shows this type of access 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 simple loops with static loop bounds. Figure 7.1b 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 extended 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

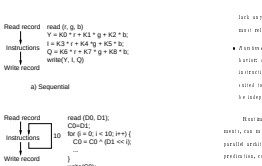

Figure 7.1: Kernel control behavior.

lack any branching support (like some graphics fragment processors) must rely on complete unrolling to execute such loops.

- *Runtime loop bounds:* Figure 7.1c shows the most dynamic of control behavior: data-dependent branching. Such kernels would require masking on instructions to execute on vector and SIMD machines, and are ideally suited to fine-grain 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 examples more complex. Data-parallel architectures have traditionally implemented conditionals by using predication, conditional streams [15], or vector masks [30]. Finer partitioning of control, such as provided by a fine-grain MIMD architecture can reduce or eliminate these overheads that conditionals have in highly synchronized architectures.

7.2.2 Benchmark attributes

Table 7.1 provides a suite of DLP kernels selected from their major application domains. Tables 7.2 and 7.3 characterize these kernels according to the computation, memory and control criteria presented previously. The two computation columns list the number of instructions and instruct ILP within the kernel (ILP is the number of instructions in an internal data kernel, divided by the dataflow graph height; when the loop bound was variable, the

Benchmark	Description
convert	RGB to YIQ conversion
dct	8 x 8 DCT of 16 bit integer block.
highpassfilter	A 3x3 high pass filter.
fft	Vorbis FFT windowing, cascade (2048 fp windows)
imdct	Vorbis imdct core.
filter3d	Filter out 16bit pixels encryption.
fir2dim	A 7x7 finite memory filter.
FFT	1024-point complex FFT.
DCT-Decompressing	DCT decompression of a data (1024x1024 matrix).
A two-triangle processing, Scale.	
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-skinning	Vertex shader for a reflection effect.
fragment-reflection	Fragment shader rendering reflected environment using cube maps.
vertex-skinning	A vertex shader used for animating a mesh with eight transformation matrices.
anisotropic-filtering	An fragment shader implementing anisotropic texture filtering [28].

Table 7.2: Benchmark attributes.

Benchmark	Memory			
	Record size words read/written	% Irregular memory accesses	% Constants	% Indexed scalar accesses
convert	3/3	-	9	-
dct	64/64	-	16	-
highpassfilter	9/1	-	9	-
fft	6/4	-	8	-
ifft	2/1	-	6	-
mp3	36/2	-	63	-
blowfish	1/1	-	2	256
rijndael	2/2	-	18	1024
vertex-simple	7/8	-	32	-
fragment-simple	8/4	-	18	-
vertex-skinning	9/3	-	35	-
fragment-reflection	5/3	-	7	-
vertex-skinning	8/9	-	32	200
anisotropic-filter	9/1	≤ 16	8	128

Table 7.3: Benchmark attributes.

- *Register accesses:* All accesses to registers are included in this group: reading, writing, register renaming, delays in router segments from the register file to a consumer, and delays in router hops in the register file. We analyze register accesses as a separate category because DLP programs often access the register file separately to read runtime constants. Since this is a read only access, it provides an opportunity for optimizations, since the register files are designed for the common case of the operand being read and written across clusters.
- *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 in control and misses and values in the data tiles, and delays in remote values back to consumers for loads. In the quantitative analysis we do not classify the memory access in to the four categories presented in Section 7.2. 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.

7.3.2 Analysis

Table 7.4 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 contributions in the critical path from fetch, register file accesses, and memory accesses, and the last column shows maximum speedup possible

Fetch: Columns two shows that on average, the instruction fetch related events account for close to 30% of the program cycles. For programs like convert, where the compiler is able to produce only small blocks it has minimal on average1, more than 95% of the program cycles are devoted to managing instruction fetch. By examining the programs current code and analyzing programs behavior we determined that the objdef provides an opportunity for concurrency

Benchmark	Percentage contribution to critical path			Speedup
	Fetch	Register access	Memory access	
DCT-convert	30.6	6.7	31.8 (DC)	6.9
DCT-dct	40.9	6.2	33.3 (DC)	7.3
IDCT-mp3-synthesis	29.4	8.7	29.3 (DC)	5.8
graphics-fragment-reflection	13.8	9.4	5.1 (L1)	2.3
graphics-fragment-simple	25.1	8.2	15.5 (L2)	5.4
graphics-vertex-skinning	13.7	9.3	25.1 (DC)	3.4
graphics-vertex-simple	21.4	9.5	23.6 (DC)	4.3
graphics-texturemapping	23.8	6.7	8.5 (32.6%)	7.5
kernels-fir2dim	27.7	20.2	15.7 (27.4%)	5.8
network-md5	37.1	5.3	5.2 (6.7%)	10.3
network-rijndael	31.2	4.2	5.0 (6.3%)	11.2
network-rtr	27.7	6.4	5.4 (10.3%)	10.3
network-reed	6.5	6.1	5.3 (72.3%)	20.7
Average	29.3	6.9	24.3	7.1

Table 7.4: Critical path analysis.

on the TRIPS architecture if all microarchitecture overheads are removed. The second row in parenthesis in the fourth column, shows the percentage of speedup network critical cycle spent in managing operands and addresses from and to the data caches.

Fetch: Columns two shows that on average, the instruction fetch related events account for close to 30% of the program cycles. For programs like convert, where the compiler is able to produce only small blocks it has minimal on average1, more than 95% of the program cycles are devoted to managing instruction fetch. By examining the programs current code and analyzing programs behavior we determined that the objdef 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-grain or MIMD execution substrate.

Register accesses: The average contribution of register accesses to the program execution is only 8.4%, but ranges from less than 2% in more than 25% as shown in the third column. As expected, programs with few operations on registers, see little of their critical path devoted to register accesses. For example fft and ifft are dominated by memory accesses and their register accesses contribution are less than 4. Register accesses become a bottleneck for applications that use a large number of architecture registers, 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 numbers fraction spent in memory. This delay includes the memories delays at the memory and the banks to reach the data in the cache banks, and memory contention delays while entering replies back to the consumers, instruction cache access delays, TLB lookups and load-to-use effect described above.

We can see a correlation between the number of memory accesses in instruction mix presented in Table 7.3 and the fraction of critical cycles contributed to by memory accesses. Mem-bnd, rijndael, network-reed-mm, fft, and JU 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. Conversely only the memory access contribution in the critical path varies from 6.8% to over 75%. Performance for programs with predominantly streamed memory accesses like fft and JU, significant part of the operand network delays are spent in inserting values in and from the memory systems, 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 7.4 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-entire instruction window, 8K-windows, and of 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 in the software model or programming model.

The speedup values range from 2.3X to almost 31X, in dicating 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 scaling baselines. As mentioned in Chapter 5, for many applications the TRIPS processor is up to 2X better than a dual-processor out-of-order superscalar processor like the Alpha 21264.

7.3.3 Summary

The quantitative analysis and the detailed program characterization showed that DLP programs shows a set of common attributes. The quantitative analysis shows that building more architecture mechanisms targeted at these specific attributes can provide significant improvements. For example, if we relaxed all of the fetch constraints for *FFT*, a 4X improvement in performance is possible. A 4X improvement in performance is possible for *LU* if all the overhead in memory accesses are removed. Secondly, since this is an analysis based on the critical path of unconstrained execution 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 modeling microarchitecture overheads. For example, some programs will fine grain communicate, can be dramatically speeded up by using decoupled execution between "threads" that the MIMD paradigm provides.

7.4 Data-Parallel Microarchitectural Mechanisms

The programmability presented in Section 7.2 provided us with insight into programs behavior and the critical path analysis in the previous section quantified the bottlenecks in the execution core, instruction control, and memory systems. In this section we describe the microarchitecture mechanisms we developed based on these insights. Figure 7.5 shows a block diagram of an abstract microarchitecture. We explain the polymorphous mechanisms in terms

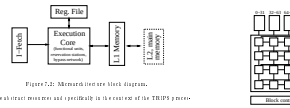

Figure 7.5: Microarchitecture block diagram.

of three sub-structures and specifically in the context of the TRIPS processor. The mechanisms are not implemented in the TRIPS prototype chip and this study where we explore what mechanisms could possibly be implemented.

7.4.1 Memory system mechanisms

The memory system in a data-parallel architecture must support high bandwidth regular memory access and low latency irregular memory accesses. One microarchitecture bottleneck analysis showed that memory accesses on average account for 38% of the critical path and optimized mechanisms could potentially produce speedups up to 8X 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 7.3 shows the configuration of the memory system that provides a high-bandwidth access through banked memories. Portions of the second-level cache banks can be reconfigured as a fully-software managed cache (SMC), in this configuration, the hardware replacement

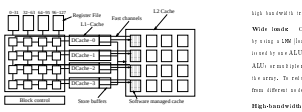

Figure 7.3: Memory systems mechanisms. Software managed cache, first class units and store buffers.

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 mapped to and are fully managed by the programmer or compiler. Only the regular memory accesses statically identifiable by the compiler are in the SMC, and they also bypass the L2 cache since temporal locality is poor. Using the data tiles which forms the L2 cache is also possible to ensure ensuring coherency at that level becomes a challenge. The programming abstraction and interfaces used in Imagine's Streams Register File (SRF) [93] may be used to manage this SMC. Providing such software managed cache (referred to as a stream register file or SRF) is a natural reconfiguration in regular memory systems. Portions of the second-level cache banks can be reconfigured as a fully-software managed cache (SMC), in this configuration, the hardware replacement

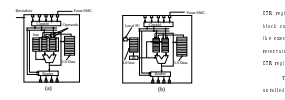

Figure 7.4: Execution core and control mechanisms. a) Instruction, operand revitalization and L0-store example. b) Local PC and L0-store mechanisms to provide MIMD execution.

consecutive iterations residing from a local storage. To efficiently support data dependent branching, each ALU is augmented with a local program counter (PC).

Local program counters: To support fine grain data dependent branching, the execution core is configured as a MIMD processing array by adding local PCs at the ALUs. To simplify the datapath wires and as a separate 3.0 micron wide storage from which instructions are fetched and executed sequentially. (A slightly more complex, but area efficient implementation is to reuse 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 storage 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

high bandwidth transfer from main memory into the SRF.

Wide inode: Overhead and latency in 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 around the reservation stations in the same ALUs in a single wave in the array. To ensure the wide port precises, a stream buffer mechanism stores these different inodes together before writing them back in 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 these 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 fine-tune latency access [94].

7.4.2 Instruction Fetch and Control Mechanisms

The branching behavior of data-parallel kernels dictate the need for fetch and control mechanisms which are: (i) pipelined fetching and mapping of revitalized instructions in reservation stations, resulting in instruction cache power and dynamic code size power, and (ii) MIMD processing support for kernels that exhibit fine grain data dependent branching. To avoid repeatedly fetching instructions of a loop, the ALUs are enhanced to revitalize instructions for

Figure 7.4: Execution core and control mechanisms. a) Instruction, operand revitalization and L0-store example. b) Local PC and L0-store mechanisms to provide MIMD execution.

consecutive iterations residing 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 revisiting instruction mappings for consecutive 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 special instruction providing the revitalizing bounds of the kernel which is stored in a special and wave control register (CR). 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

CR 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 causes the instructions at the instructions in the reservations stations, pinning them for executing another iteration. When the CR register reaches zero, the local 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 reservation stations, so as to reduce the number of revitalizations. Figure 7.4a shows the datapath and control path modifications added by this mechanism. The shaded regions need to the reservation stations indicate the state bits required for revitalization. In the TRIPS processor, using instruction revitalization provides a nearly SIMD-like architecture model.

Local program counters: To support fine grain data dependent branching, the execution core is configured as a MIMD processing array by adding local PCs at the ALUs. To simplify the datapath wires and as a separate 3.0 micron wide storage from which instructions are fetched and executed sequentially. (A slightly more complex, but area efficient implementation is to reuse 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 storage 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

ATTRIBUTE	Mechanisms	Implemented at	Benchmarks that benefit
Regular memory access	Software managed L2-memory cache,	L2-Memory	ALL
Irregular memory access	Cached memory cache	L1-Memory	Vertex+GEOM, Fragment+GEOM
Scalar constant	Local operand storage, Operand revitalization	Reservation core, Register file	convert, Ct, Highpass filter, mult, clustered, all quality programs
Indexed named constant	Software managed L0 data store at ALU,	Reservation core	blocksh, cjpeg, rijndael, encryption
Tight loops	Instruction revitalization, Instruction revitalization	Reservation core, Instruction fetch	All
Make dependent branching	Local program counters, local PC	Instruction fetch,	vertex+shading, rasterization+Shading

Table 7.3: Data-parallel program attributes and the set of universal microarchitectural mechanisms. Mechanisms in parenthesis indicate TRIPS specific implementations.

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: (i) a software managed streamed memory subsystem is used to support high bandwidth regular memory accesses, and (ii) a hardware managed cache memory subsystem is used to support efficient irregular memory accesses. The execution core is enhanced with additional local operand storage to efficiently support named constant operands, 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 cores is added for supporting at most 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, cleverly different and the L0 instructions can be added in a straightforward manner to conventional write-once combined or clustered superscalar architectures by adding direct channels from the L2-cache to the functional units and augmenting the problem to memory instructions dependent on the bank where their operands come from the SMC. The Tarantula architecture provides similar cache support for transfers from the L2 memory in the vector register file, using hardware techniques to generate conflict free addresses to different banks in memory, in contrast to our approach of pushing all the regular accesses in a single bank. To support indexed constants and irregular memory accesses in this architecture, the L0-scache memory must be addressable using special vector/gather instructions. Also conventional superscalar processors provide good support for L1-scache memories.

The reservation stations in TRIPS have a many-to-one current and one to reservations stations in superscalar architectures and both the instruction and operand revitalization mechanisms can be applied to provide instruction and operand revitalization. Many DSP processors have implemented zero-overhead branches in different ways to support tight loops.

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 streams and thus must be augmented with a local PC and storage buffers to provide a MIMD model

Benchmark	Ops/cycle	Benchmark	Ops/cycle
convert	7.3	fragment-simple-interpolated	5.8
dct	8.4	fragment-simple-interpolated-noperspective	6.7
highpassfilter	5.9	fragment-reflection	5.3
fft	6.3	fragment-simple-noperspective	5.3
lu	6.2	fragment-reflection-noperspective	5.4
md5	5.8		
blowfish	5.2		
rijndael	5.9		

Table 7.4: Performance on baseline TRIPS.

integer multiplies, and an FPU with add, multiply, and divide capability.

Table 7.4 shows the performance of the baseline measured in terms of number of useful computation instructions executed per cycle, not including overhead instructions like address computation and load store instructions. Only the DSP programs contain a very high computation throughput, averaging about 3 ops/cycle, while all other applications sustain a low three digit ops, averaging about 2 ops/cycle.

Since the baseline TRIPS processor is optimized for ILP, recovering the data level parallelisms in these applications in ILP results in inefficiencies for DLP programs. For example, loop-counter is redundantly sunrolled in parallel large enough blocks is 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 DSP programs have large demands on these resources, the limited bandwidth prevents the architectures from achieving its potential performance.

Config	L0 store		Mem bandwidth	Architecture needed
	Scope	Size/ALU	Bytes/cycle	
S	No	No	5	SMIMD
S-O	Yes	No	Yes	SIMD+vector-constant-ops
S-O-D	Yes	Yes	Yes	SIMD+vector-constant-ops+Software-managed cache
M	Yes	No	No	MIMD
M-D	Yes	Yes	No	MIMD+memory hierarchy

Table 7.2: Machine configurations.

7.5.3 Configuration of Mechanisms

The mechanisms described in Section 7.4 can be combined in different ways according to application requirements to produce as many as 20 different machine configurations. Figure 7.6 shows some 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 7.2 and 7.3 determine the ideal combination of mechanisms on the TRIPS processor. In this dissertation we focus on three node configurations, shown in Table 7.2, that cover the applications we examined.

In all the configurations, one memory bank per row is configured to be used as a software managed cache. The SMC banks store 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 mechanisms creates a baseline model that is similar to SIMD and vector machines. Instruction revitalizations adds the support for instruction and control efficiency that make SIMD and vector machines efficient at DLP. The reservations stations distributed across the tiles can be built of performing a distributed vector register file and the instructions mapped across the different tiles forms one large vector instructions.
- **SIMD + scalar operand access:** This baseline machine (S) can be augmented with operand revitalization to create the S-O machine. This configuration optimizes the execution of kernels.
- **SIMD + scalar operand + lookup table access:** The S-O-D machine adds local 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 executing in pure dataflow fashion.
- **MIMD + lookup table access:** Addition of local L0 data storage creates a premier configuration creates the M-D machine.

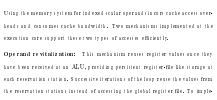

Figure 7.6: Speedup using different mechanisms, relative to baseline architecture. Programs grouped by best machine configuration.

7.5.4 Performance Evaluation

Figure 7.6 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 configurations.

- **SIMD extensions (S):** gfl and h.264 are vector-oriented benchmarks and require high memory bandwidth and high instruction fetch rate. Compared to the baseline a five-fold speedup is achieved because of the higher ALU utilization and higher memory bandwidth of the S configurations.

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 (25 constants in vertex effects in the example). These performs best on the S-O machine configuration as shown by the bars of 2 programs in Figure 7.3.
- **SIMD + scalar operand + lookup table access (S-O-D):** fft, lu, and rijndael which are enormously large lookup tables show speedups of 27% and 88% respectively over the S-O configuration, but performs poorer than the M-D machine.
- **MIMD (M):** The baseline MIMD configuration degrades performance compared to architecture S-O-D for all applications except vertex+shading. 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+shading. With local looping control, these programs require far less instruction storage and hence can be enrolled more aggressively providing more parallelism. Because vertex+shading uses data dependent branching, the overheads of predicated execution (normalization) are also removed.
- **Flexibility:** The last single bar labeled *Flexible* in Figure 7.3 is where the harmonic mean of speedup achieved by a flexible architecture when a subset of mechanisms are combined according to application needs (missing gfl and h.264 as current through vertex simple light as S-O, and the rest as M-D). Averaged across the different applications, this flexible dynamic using provides 55% better performance over a fixed S configuration, 20% better than fixed S-O and 5% better than a fixed M-D machine.

7.5.5 Comparison against specialized architectures

Table 7.5 shows the results of a rough comparison between the performance of the configurable TRIPS architecture to published performance results on specialized hardware. Column 2 and 3 show performance, column 4 shows the performance metric (which runs), and column 5 describes the specialized hardware. For each of the applications we picked the best combination of the mechanisms on the TRIPS hardware. 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 prototype

Benchmark	Performance		Units	Reference/how
	TRIPS (clock rate scaled)	Hardware/ ISA or Software		
	DSP kernels			
convert	4.64	0.7	Recalculations/ second/cycle	DSP-C 6416 (1GHz DSP processor)
highpassfilter	9.3	5.3	Images/cycle	Imagine [93] (multimedia processor)
fft	4.64	1.5	Recalculations/ second/cycle	Tarantula [95] (vector unit)
lu	4.64	1.5	Recalculations/ second/cycle	Tarantula [95] (vector unit)
Network processing kernels				
md5	4.5	3	cycles/block	Cryptomaniac [96]
blowfish	8	18	cycles/block	
rijndael	12	24	cycles/block	
Graphics processing kernels				
			hardware	
fragment-simple-interpolated	84	2100	fragments/ second/cycle	NVIDIA GeForce FX 4300(fragment processor)
fragment-simple-interpolated-noperspective	102		fragments/ second/cycle	
fragment-reflection	48	24	fragments/ second/cycle	
fragment-reflection-noperspective	48		fragments/ second/cycle	

Table 7.5: Performance comparison of TRIPS with DLP mechanisms to specialized hardware.

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 its use of microarchitecture networks for communication across modules.

Polymorphism: This dissertation introduces architectural polymorphism: the capability in processing hardware across coarse-grain partitions different functions. Unlike reconfigurable architectures that synthesize complex logic from primitive functions, the polymorphous principle is to build coarse grain reconfigurable microarchitectural blocks whose function can be changed at runtime. We used the TRIPS architecture as the baseline for developing and implementing the polymorphous mechanisms. The TRIPS architecture is a modular design with well-defined microarchitecture blocks and a technology scalable design, thereby serving as a good baseline starting point for implementing polymorphisms. We proposed and evaluated mechanisms targeted at three processor scenarios: 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 inherent 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 3 to 4. On the EEMBC and SPEC CPU2000 benchmarks, with compiler-generated code we see IPCs in the range of 0.2 to 2.2, with an average IPC of 1.1 for the EEMBC suite and 1.4 for the SPEC CPU2000 suite. On hand optimized microkernels, the TRIPS processor is up to 4 times better than an Alpha 21264. With compiler generated code for the 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 9 times better than the Alpha 21264 and many benchmarks show several of the same optimizations. Some of these hand optimizations, which include better instruction merging, local store dependence elimination through better register allocation, and similar instructions level optimizations producing code with compilation tweaking, are not unreasonable to implement in the compiler. These are currently hand optimization and are yet in the way for the 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 bottlenecks behaviors as compilers deliver versions we have clients. As the compiler matures and we develop a better understanding of the heuristics, we expect more of these optimizations to be integrated into our compilers 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 parallelisms. When executing 4 threads on a single processor, significantly higher levels of processor utilization are seen, IPCs are in the range of 8.2 to 8.5 for an application mix consisting of EEMBC and SPEC CPU2000 workloads. Compared to an average IPC of 5.3 and 5.6, these application mixes have much higher IPCs - 2 when running with 2 application mix concurrently, and 3.5 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 5.8 across a set of DLP workloads. The speedup provided by the individual mechanisms range from 3 to 15.2. The polymorphous mechanisms enable the TRIPS architecture to match the performance of specialized processors targeted at different types of DLP workloads. Synthesizing the polymorphous mechanisms allow the configurable TRIPS chip to match the performance of best-of-breed DSP chips, graphics chips, and router chips on workloads specialized for each.

8.2 Discussion

We have developed a prototype chip that implements the TRIPS ISA and at the time of this dissertation, we expect systems bank at the end of Fall 2006. In 2003 we started with prototyping results based on high level simulations. The implementation of the prototype chosen that these ideas are feasible, and the microarchitecture networks chosen that a block atomic model

We programmed the graphics kernels for the NVIDIA GeForceFX chip and measured performance on a 2.4 GHz Pentium-based system. In the vertex-simple graphics applications, 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 vs floating point units) on the GeforceFX are not publicly disclosed, we believe part of this high performance can be attributed to the larger number of execution units. The vertex-graphics processing kernels are more complex (using more instructions, more constants, and data dependent branching in one case) than the ones we benchmarked, and will perform at best as well as the hardware based, and likely poorer.

7.6 Summary

In this chapter we presented a comprehensive treatment of programs covering a large spectrum of the DLP applications space, including signal processing, scientific, network/security, and emerging graphics applications. While there may be DLP applications outside these domains, the kernels 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 architectural polymorphous 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-2 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 to the application as simulations is the philosophy of next-generation processors [52], 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, while 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 conclusion that we propose several strictly limited in the TRIPS processor described in this dissertation. For example the hybrid of SIMD and fine-grain MIMD execution models is a reasonable goal for other DLP architectures. Future systems that must execute multiple classes of DLP applications well needs to implementing all of the mechanisms and dynamically configu-

ring 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 as a fixed architecture by matching the mechanisms to the application attributes.

Chapter 8
Conclusions

Processor architects today are faced by two daunting challenges: emerging applications with heterogeneous computation needs and technology limitations of power, wire-delays, and process variations. Designing multiple applications specific processors or specialized architectures introduces design complexity, software programmability problems, and reduces economies of scale. There is a pressing need for some design philosophy that can provide support for heterogeneous applications, combat processor complexity, and achieve 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 parallelisms. The basic idea in polymorphism is to configure coarse grain microarchitecture blocks to provide an adaptive and flexible processor architecture. Technology scalability is achieved with scalable and modular microarchitecture blocks.

8.1 Summary

In this dissertation, we identified the granularity of parallelisms as the fundamental differentiators between application classes and used granularity-

cation heterogeneity with respect to processor architectures. The three granularities of parallelisms are instruction level parallelism, thread level parallelism, and data level parallelism. To provide architectural support across all these type of parallelisms, we propose architectural polymorphism driven by three main principles: adaptively across these granularities of parallelisms, economy of mechanisms, and microarchitectural reconfiguration at a coarse granularity.

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

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 parallelisms. 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 changes at an instruction level which leads to a lot of instruction level overhead, architecture blocks operates 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 its use of microarchitecture networks for communication across modules.

Polymorphism: This dissertation introduces architectural polymorphism: the capability in processing hardware across coarse-grain partitions different functions. Unlike reconfigurable architectures that synthesize complex logic from primitive functions, the polymorphous principle is to build coarse grain reconfigurable microarchitectural blocks whose function can be changed at runtime. We used the TRIPS architecture as the baseline for developing and implementing the polymorphous mechanisms. The TRIPS architecture is a modular design with well-defined microarchitecture blocks and a technology scalable design, thereby serving as a good baseline starting point for implementing polymorphisms. We proposed and evaluated mechanisms targeted at three processor scenarios: 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 inherent 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

can be effectively implemented by a physically distributed design.

These distributed protocols have enabled us to construct a 16-wide, 1024-in-flight, speculative, out-of-order processor, which while 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 heterogeneous computing substrate in 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 systems 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.

8.2 Discussion

We have developed a prototype chip that implements the TRIPS ISA and at the time of this dissertation, we expect systems bank at the end of Fall 2006. In 2003 we started with prototyping results based on high level simulations. The implementation of the prototype chosen that these ideas are feasible, and the microarchitecture networks chosen that a block atomic model

a detailed understanding of program behavior and quantitative analysis. For example, we demonstrated a clear intuition 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 revitalization events that enabled fetched instructions to be reused in the processor core without instructions and new instruction sequences. The number of mechanisms in core ILP, TLP, and DLP are few in number and implementing these would be simpler than building multiple cores on chip.

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 [95], whereas the polymorphous TRIPS design includes two homogeneously polymorphous processor cores. The synthetic 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 new vector unit at significant design cost, Tarantula provides global synchronization between the different vector lanes with partitioned vector registers and optimized access

moves 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 completed the memory system and instead provide support to create a software managed memory system by reconfiguring the L2 cache banks as a vertical memories. While the Tarantula approach allow vector access in the L2 cache address a complex conflict free address generation scheme to maximize bandwidth [95], to create a vertical memories at the banks level in interesting queries in address wide decoding are which solution to pick. While this dissertation leaves the question open, we do see there is an interesting area to address.

- While Tarantula which contains vector register files which need to be read and written for every instruction, we showed that did not duplicate it in the primary chip; polymorphous mechanisms that can use the memories at various levels elegantly and with each ALU to create vector register file like behavior with register bypassing capability.
- Since Tarantula is a vector processing core, accesses to the L2 caches are disabled, consequently programs which require lookup tables, large number of constants and other irregular data structures perform poorly. In the TRIPS approach, an application can decide 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 how how to limit the interactions between these mechanisms and thus achieve verification closure. The mechanisms are by definition localized and can be used separately or together. For example, the five DLP mechanisms work in about 20 process configurations which presents a reasonably daunting verification challenge. With a heterogeneous solution, the number of replicated 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 with modeling and which solution to pick. While this dissertation leaves the question open, we do see there is an interesting area to address.

TRIPS prototype chip implements a limited amount of such polymorphous support where the mechanisms can be dynamically shares, for example, the multi-threaded mode of the processor, a single-block execution mode of the processor, and the configuration of the memory like a scratchpad memories. We verified these mechanisms and modes of the processor through randomized testing by generating random programs and checking on the processor modes through simulations. The level of coverage achieved in this process leads us to believe that the verifications is not much more difficult than verifying multiple heterogeneous cores.

8.2.2 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 [55]. Other recent examples include, FLAME [59] and ATLAS [161] targeted at linear algebra, SPIRAL [120] which uses a dynamic programming approach to optimize the compilation of DSP routines, and the Broadway compiler meant for scientific computing libraries [61]. Programming language efforts include, Streamit [57] targeted at streaming and multimedia programs, Cg targeted at graphics processing [103], Shangri-La targeted at network processing [30], and a high-level specification system for quantum chemistry computations that can generate optimized parallel code [19].

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 IBM 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 [43]. With growing heterogeneous application needs and the increasing capability 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.

8.2.3 Technology constraints

This dissertation has focussed 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.

8.3 A Brave New World of Polymorphous Architectures

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 expose 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 for other technology objectives. We have focused on polymorphism to improve performance. These principles of polymorphism can be used for other objectives like: 1) achieving different levels of power efficiency, 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. One application of the ideas in this dissertation is to use the principles here to determine what mechanisms are required when the need is to support only a specific set of applications. 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 polymorphous, ISAs with block atomic execution with concurrency explicity 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 [28]. 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. Because this models the hardware at such a detailed level it is not very extensible and we used a more abstract simulator called the GPA simulator for our DLP study in chapter 7. This simulator uses binaries generated by the IMPACT compiler and 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 the two.

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 [159], 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 modelling 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.

A.2 Results

Table A.1 shows the comparison of the two simulators on the DLP kernels used in the DLP study in chapter 7. 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 modelling 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,

Benchmark	GPA simulator		TRIPS simulator		Ratio	
	Cycles	Insts	Cycles	Insts	Cycles (T/G)	Insts (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
Mean	313921	893002	664487	1177700	2.1	1.4

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

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, potentially creates a situation in which the difference between the two simulation environments is primarily the microarchitecture modelling. Now, on average, the GPA simulator over-estimates performance by 1.4X.

The results from these two controlled experiments, lead us to conclude

Benchmark	GPA simulator		TRIPS simulator		Ratio	
	Cycles	Insts	Cycles	Insts	Cycles (T/G)	Insts (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
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

that the compiler quality and the modelling 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.

Appendix B

IPC reduction from speculation depth

This appendix includes a performance comparison of the ILP mode to the 1-Thread 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.

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 with 8-deep and 2-deep speculation - 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 with 8-deep and 2-deep speculation - 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 with 8-deep and 2-deep speculation - DLP suite

Bibliography

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

[2] GPGPU: www.gpgpu.org.

[3] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: architecture and performance. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 2–13, 1995.

[4] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug 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] Kurt Akeley. Reality engine graphics. In *Proc. of the 20th Ann. Conf. on Computer Graphics*, pages 109–116, 1993.

[6] David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley E. Schuster. Dynami-

cally tuning processor resources with adaptive processing. *Computer*, 36(12):49–58, 2003.

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

[8] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, 1990.

[9] Jeff Andrews and Nick Baker. Xbox 360 system architecture. *IEEE Micro*, 26(2):25–37, 2006.

[10] Marco Annaratone, Emmanuel A. Arnould, Thomas Gross, H. T. Kung, Monica S. Lam, Onat Menzilcioglu, and Jon A. Webb. The Warp Computer: Architecture, Implementation, and Performance. 36(4):1523–1538, December 1987.

[11] Arvind and David E. Culler. Dataflow architectures. 1:225–253, 1986.

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

[13] A. Arvind and K.P. Gostelow. The U-Interpreter, 1982.

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

[15] Saisanthosh Balakrishnan and Gurindar 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, 2006.

[16] Max Baron. MP Cores for Handheld Apps. *Microprocessor Report*, December 2005.

[17] L.A. 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 International Symposium on Computer Architecture*, pages 282–293, June 2000.

[18] 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.

[19] G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R.M. 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.

[20] Tom Blank. The Maspar MP-1 architecture. In *Proc. of IEEE Compcon, Spring 1990*, pages 20–24.

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

[22] 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 '88*, pages 330–339, November 1988.

[23] Shekhar Y. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[24] 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.

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

[26] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream

computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

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

[28] Doug Burger and Todd M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[29] Doug Burger et al. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[30] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. Shangri-La: achieving high performance from compiled network applications while enabling ease of programming. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 224–236. ACM Press, 2005.

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

[32] K. Coons, X. Chen, S. Kushwaha, K. S. McKinley, and D. Burger. A Spatial Path Scheduling Algorithm for EDGE Architectures. In *Proceed-*

ings of the Twelth International Conference on Architectural Support for Programmin Languages and Operating Systems (ASPLOS), 2006.

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

[34] NVIDIA Corp. NVIDIA GPU programming guide, v2.2.1, Nov.

[35] Adrian Cristal, Oliverio J. Santana, Mateo Valero, and Jose F. Martinez. Toward kilo-instruction processors. *ACM Transactions on Architecture and Code Optimization*, 1(4):389–417, December 2004.

[36] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John 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.

[37] William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, Franois Labont, Jung Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, and Ian Buck. Merrimac: Supercomputing with Streams. In *The Proceeding of the 2003 International Conference for High Performance Computing, Networking, Storage, and Analysis SC 03*, November 2003.

[38] 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.

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

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

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

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

[43] A. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden 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.

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

[45] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isacc Hernandez, Toni Juan, Geoff Lowney, Ma thew Mattina, and Andre Seznec. Tarantula: A vector extension to the alpha architecture. In *Proceedings of The 29th International Symposium on Computer Architecture*, pages 281–292, May 2002.

[46] Roger Espasa, Mateo Valero, and J. E. Smith. Out of order vector architectures. In *Proc. of the 30th Ann. Int'l Symp. on Microarchitecture*, pages 160–170, December 1997.

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

[48] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. Addison-Wesley Publishing Company, 2003.

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

[50] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 146–156, 1995.

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

[52] Joseph A. Fisher, Paolo Faraboschi, and Giuseppe Desoli. Custom-fit processors: Letting applications define architectures. In *Proc. of the 29th Ann. Int'l Symp. on Microarchitecture*, pages 324–335, December 1996.

[53] M Flynn. Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 948, 1972.

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

[55] Matteo Frigo. A fast Fourier transform compiler. In *PLDI 1999: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 169–180. ACM Press, 1999.

[56] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and Reed Taylor. Piperench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4):70–77, April 2000.

[57] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Hoff-

man, David Z. Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, October 2002.

[58] Paul Gratz, Changkyu Kim, Robert McDonald, Stephen W. Keckler, and Doug Burger. Implementation and Evaluation of On-Chip Network Architectures. In *Proceedings of the 24th International Conference on Computer Design*, pages 170–177, October 2006.

[59] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, 2001.

[60] Rajiv 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.

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

[62] Ziyad S. Hakura and Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proc. of the 24th Int'l Symp. on Computer Architecture*, pages 108–120, June 1997.

[63] Tom R. Halfhill. IDF Delivers Extreme Surprises, October 2003.

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

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

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

[67] Tom R. Halfill. MathStar Challenges FPGAs. *Microprocessor Report*, July 2006.

[68] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, 1997.

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

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

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

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

[73] John R. Hauser and John 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.

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

[75] John Hennesy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, 1996.

[76] H. Peter 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.

[77] M. S. Hrishikesh, Doug Burger, Stephen W. Keckler, Premkishore Shivakumar, Norman P. Jouppi, and Keith 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.

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

[79] Quinn Jacobson, Steve Bennett, Nikhil Sharma, and James E. Smith. Control flow speculation in multiscalar processors. In *Proceedings of the*

3rd International Symposium on High Performance Computer Architecture, pages 218–229, February 1997.

[80] 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.

[81] 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.

[82] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient Conditional Operations for Data-parallel Architectures. In *Proc. of the 33rd Ann. Int'l Symp. on Microarchitecture*, pages 159–170, 2000.

[83] Ujval J. Kapasi, Peter Mattson, William J. Dally, John D. Owens, , and Brian Towles. Stream Scheduling. In *Proceedings of the 3rd Workshop on Media and Streaming Processors*, pages 101–106, December 2001.

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

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

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

[87] Ho-Seop Kim and James E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *ISCA*, pages 71–80, 2002.

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

[89] 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.

[90] Christos Kozyrakis and David Patterson. Overcoming the limitations of conventional vector processors. In *Proc. of the 30th Int'l Symp. on Computer Architecture*, pages 399–409, June 2003.

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

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

[93] Kevin Krewell. Startup Aegia Accelerate Reality. *Microprocessor Report*, April 2005.

[94] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 81–92, 2003.

[95] Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen. Conjoined-core chip multiprocessing. In *MICRO*, pages 195–206, 2004.

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

[97] L.E. Shar and E.S. Davidson. A Multiminiprocessor System Implemented Through Pipelining, Computer, Feb. 1974, pp.42-51.

[98] Ruby B. Lee. Subword parallelism with max-2. *IEEE Micro*, 16(4):51–59, 1996.

[99] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural*

support for programming languages and operating systems, pages 46–57, New York, NY, USA, 1998. ACM Press.

[100] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. 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.

[101] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 161–171, June 2000.

[102] William R. Mark and Donald 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.

[103] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Proc. of the 30th Ann. Conf. on Computer Graphics*, 2003.

[104] Robert McDonald, Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen 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.

[235] B. Moore, A. Padegs, R. Smith, and W. Bucholz. Concepts of the System/370 Architecture. In *Proc. of the 1st Ann. Symp. on Computer Architecture*, pages 282–292, June 1997.

[236] Charles R. Moore. Managing the Transition from Complexity to Elegance: Design Convergence. *IEEE Micro*, 24(1):8–23, 2004.

[237] Ramadass Nagarajan. *Design and Analysis of Technology-Scalable Architectures*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, December 2006.

[238] Ramadass Nagarajan, Xia Chen, Robert G. McDonald, Doug Burger, and Stephen W. Keckler. Critical path analysis of the trips architecture. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 37–47, March 2006.

[239] Ramadass Nagarajan, Sundeep K. Kushwaha, Doug Burger, Kathryn S. McKinley, Calvin Lin, and Stephen W. Keckler. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In *13th International Conference on Parallel Architecture and Compilation Techniques*, pages 74–84, October 2004.

[240] Ramadass Nagarajan, Karthikeyan Sankaralingam, Stephen W. Keckler, and Doug Burger. A Design Space Evaluation of Grid Processor Architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 40–51, December 2001.

[241] Parmesh S. Oberoi and Gurindar S. Sohi. Parallelism in the front-end. In *ISCA*, pages 230–240, 2003.

[242] John Oliver, Ravishankar Rao, Paul Sultana, Jedidiah Crandall, Erik Czernikowski, Leslie Jones IV, Diana Franklin, Venkatesh Akella, and Frederic T. Chong. Synchroscalar: A Multiple Clock Domain, Power-Aware, Tile-Based Embedded Processor. In *Proceedings of the 31th Annual International Symposium on Architecture*, pages 150–161, June 2004.

[243] Tim Olson. Advanced Processing Techniques Using the Infinity Pre-MATH Processor. In *Embedded Processor Forum*, May 2002.

[244] Alex Pajuelo, Antonio Gonzalez, and Mateo Valero. Speculative Dynamic Vectorization. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 37–48, May 2002.

[245] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. of the 24th Ann. Int'l Symp. on Computer Architecture*, pages 206–218, June 1997.

[246] Alex Peleg and Uri Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, 1996.

[247] 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.

[248] Matt Pharr and Greg Humphreys. Design and Implementation of a Physically-Based Rendering System. Draft edition, August, 2003.

[249] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 90–101, 2001.

[250] Matthew Poremba, Joe Meehean, Jeremy Johnson, David Padua, Mauricio Velasco, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, , and Nick Rizzolo. SPIRAL: Code Generation for DSP Transforms. 93(2):232–275, 2005.

[251] Steven E. Raasch, Nathan L. Binkert, and Steven K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 318–329, 2002.

[252] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, November 1994.

[253] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3–13, 1998.

[254] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3–13, December 1998.

[255] Brad Rodriguez and Will Moore and Andrew McCabe. Spatial Arrays. Institute of Physics Publishing, 1997.

[256] 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.

[257] Richard M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[258] A. Saavedra and R. Royuela. Confluence-free execution of a unified version on a hashed cache. *IEEE Transactions on Computers*, 54(7):824–836, 2005.

[259] H Saunders. *ALP: Energy Efficient Support for All Levels of Parallelism for Complex Media Applications*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, July 2005.

[260] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Stephen W. Keckler, Doug Burger, and Charles R. Moore. Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, page 422–433, June 2003.

[261] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M.S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 480–491, December 2006.

[262] Karthikeyan Sankaralingam, Vincent Ajay Singh, Stephen W. Keckler, and Doug C. Burger. Routed Inter-ALU Networks for ILP Scalability and Performance. In *Proceedings of the 21st International Conference on Computer Design*, pages 179–188, October 2003.

[263] Richard M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[264] A. Saavedra and R. Royuela. Confluence-free execution of a unified version on a hashed cache. *IEEE Transactions on Computers*, 54(7):824–836, 2005.

[265] H Saunders. *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.

[266] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Stephen W. Keckler, Doug Burger, and Charles R. Moore. Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, page 422–433, June 2003.

[267] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M.S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 480–491, December 2006.

[268] Karthikeyan Sankaralingam, Vincent Ajay Singh, Stephen W. Keckler, and Doug C. Burger. Routed Inter-ALU Networks for ILP Scalability and Performance. In *Proceedings of the 21st International Conference on Computer Design*, pages 179–188, October 2003.

[269] A. Saulsbury, *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.

[270] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3–13, December 1998.

[271] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3–13, December 1998.

[272] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3–13, December 1998.

[273] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M.S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 480–491, December 2006.

[274] Karthikeyan Sankaralingam, Vincent Ajay Singh, Stephen W. Keckler, and Doug C. Burger. Routed Inter-ALU Networks for ILP Scalability and Performance. In *Proceedings of the 21st International Conference on Computer Design*, pages 179–188, October 2003.

[275] A. Saulsbury, *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.

[276] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3–13, December 1998.

[277] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3–13, December 1998.

[278] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M.S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 480–491, December 2006.

[279] Karthikeyan Sankaralingam, Vincent Ajay Singh, Stephen W. Keckler, and Doug C. Burger. Routed Inter-ALU Networks for ILP Scalability and Performance. In *Proceedings of the 21st International Conference on Computer Design*, pages 179–188, October 2003.

[280] A. Saulsbury, *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.

[281] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3–13, December 1998.

[282] O. Wolf and J. Bier. StarCore Launches First Architecture. *Microprocessor Report*, 12(14):17–22, October 1998.

[283] Lisa Wu, Chris Weaver, and Todd Austin. CryptoManiac: A Fast Flexible Architecture for Secure Communication. In *Proc. of the 30th Int'l Symp. on Computer Architecture*, page 110–119, June 2003.

[284] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Proceedings of the 1st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, page 49–58, June 1995.

[285] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Tinoosh Mohsenin, Mandeep Singh, and Bevan 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.

[286] J. M. Teuhler, J. S. Dodson, Jr. J. S. Fields, B. Lu, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 26(1):5–46, January 2001.

[287] 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.

[288] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. Vis speeds new media processing. *IEEE Micro*, 16(4):10–20, 1996.

[289] Tomarus : An infrastructure for research in multicore level parallelism. http://www.tomarus.org.

[290] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture*, pages 191–202, New York, NY, USA, 1996. ACM Press.

[291] Dean M. Tullsen, Susan J. Eggen, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, 1995.

[292] Jim Turley. Texilica CPU Binds to Designers' Will. *Microprocessor Report*, 12(4), March 1998.

[293] Tim Ukpong, Rami Rokkila[0285], and Jani Kr#352pic. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(3):229–243, 2003.

[294] Sriram Vajapeyam and Tulika 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.

[295] 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.

[296] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: RAW Machines. *Computer*, 30(9):86–93, 1997.

[297] R. Clint Whaley and Jack 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.

[298] Jim Turley. Texilica CPU Binds to Designers' Will. *Microprocessor Report*, 12(4), March 1998.

[299] Tim Ukpong, Rami Rokkila[0285], and Jani Kr#352pic. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(3):229–243, 2003.

[300] Sriram Vajapeyam and Tulika 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.

[301] 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.

[302] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: RAW Machines. *Computer*, 30(9):86–93, 1997.

[303] R. Clint Whaley and Jack 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.

Vita

Karthikeyan Sankaralingam was born in Chennai, India on 6th February 1978, the son of Parthiban Sankaralingam and Alagammai Sankaranmai. He received the Bachelor of Technology degree in Aerospace Engineering from 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: 3113 Willow Creek Dr.
Apt. 203
Austin, Texas 78745

This dissertation was typeset with LaTeX[1] by the author.

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