PROCESSOR EFFICIENCY
FOR PACKET-PROCESSING APPLICATIONS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Elizabeth Seamans
September 2005

Preface

Acknowledgments

Contents

List of Tables

List of Figures

Abstract

Incorporating programmable hardware in a wide variety of network routers is a subject of current investigation in academia and industry. Small, all-software routers, for example, employ programmable hardware to serve customers at network end-points: commodity hardware such as desktop x86 platforms serve as an inexpensive solution for these low-end routers. At the other end of the spectrum, large network routers, which have historically used hardwired platforms for high performance and given each packet the same "best effort" handling, are beginning to include some customized programmable hardware to meet a growing need for specialized packet handling. Programmable hardware provides greater flexibility than hardwired platforms and allows large routers to offer packet processing with more runtime variability as well as new or updated packet services.

Large routers can benefit from added flexibility, but they must continue to support their high volume of traffic. Although a wide range of designs using programmable hardware to supply the combination of performance and flexibility for packet processing have been proposed and implemented, no dominant approach has emerged. In this dissertation we have evaluated a comprehensive set of techniques for mapping work to an array of processor cores, and several methods for managing the resources of an individual processor. We present work to show that specific characteristics of our application domain can be leveraged to increase the efficiency of programmable hardware and manage the contention for hardware resources.

We employ a concrete model of a hardware platform and an application composed of a chain of dependent tasks to demonstrate that packet-processing applications in our domain can best use a processor array by exploiting the inherent data parallelism available in the independent packets in the network traffic. We identify the four potential benefits available to the less efficient task parallel implementations and explore ways to provide three of those benefits for our data parallel implementation. We present analytical models which predict performance based on key system and

application characteristics and evaluate performance sensitivity to available hardware resources and virtualized hardware contexts.

Chapter 1

Introduction

Routers forward packets through a wide area network and frequently provide additional packet services for the packets they forward. Historically, routers which handle large volumes of packet traffic have been implemented on mostly hard wired platforms to provide homogeneous processing to high volumes of network traffic. Packets which required additional or alternate processing were siphoned off from regular network traffic and handled separately as exceptions. While this strategy was successful for relatively homogeneous packets, changes in network traffic demand more services and more heterogeneous packet handling in the routers. Customers have recently begun to request processing adapted to finer traffic granularity: for example, they may want individual packet streams to have guaranteed bandwidth or protocol-specific processing. Traditional hardwired systolic pipeline models are inflexible and have difficulty accommodating this kind of specialized handling requirements; implementing some large router packet processing services on programmable hardware, which execute software-based instructions, could provide the necessary flexibility.

Although large routers need more flexibility to apply heterogeneous processing to packets in their network traffic, their performance remains critical: at the same time, they are typically constrained by the power and area available to the chip set. Because programmable hardware uses those resources less efficiently than hardwired designs, it is likely to support slower computation than its hardwired counterpart. However, packet processing performance in large routers is frequently memory bound: the off-chip memory bandwidth, which must support frequent random accesses to off-chip data structures, is usually the system bottleneck. As long as the compute hardware, whether hardwired or programmable, can keep the memory bandwidth occupied the system can support the required throughput. The resulting challenge is to determine how to use a limited amount of programmable hardware efficiently, making the best use of the available power, chip area, and off-chip memory bandwidth, to increase the amount of flexible packet processing power available in the router.

1.1 Contributions

This dissertation makes the following contributions:

1. it presents a problem area which is outside the focus of current research, efficient flexible packet processing in high performance edge routers, and uses a concrete target platform and a widely-used packet-processing application as an example packet-processing system incorporating programmable hardware,

2. it evaluates strategies for mapping work onto multiple processors and techniques for loosening the instruction fetch bandwidth bottleneck,

3. it uses analytical models to demonstrate that although the total number of instructions per packet determine the maximum potential throughput, specific characteristics of our application domain can be exploited to increase resource utilization and manage contention for resources, including

    - the number of instructions executed between remote memory references,
    - the compact layout of persistant data objects, and
    - short, simple computation using the persistant data objects,

4. it demonstrates that exploiting the data parallelism available in independent packets, supported by virtualized hardware contexts, results in simple and efficient programmable hardware packet processing systems, and

5. it demonstrates that several benefits derived from the less efficient task parallel implementations can be provided with a data parallel implementation, including
   - scaling performance beyond the number of hardware contexts,
   - improving throughput by loosening the instruction fetch bandwidth bottleneck, and
   - reducing packet latency.

1.2 Background

We have already discussed the motivation for using programmable hardware in large network routers and the contributions of this dissertation to that end. In this section we describe the large edge router which holds the programmable hardware, our example packet-processing application we expect the programmable hardware to execute, and the flexibility we would like the system to support. Together, they represent the target system for the experiments we report in this dissertation.

1.2.1 Edge Router: High Performance with Flexibility

A large *edge router* which manages traffic at the edge of a network domain is a good example of a router providing a variety of services to high volumes of heterogeneous network traffic. Edge router architectures vary, but we define a typical edge router architecture with the following characteristics:

1. The edge router receives and transmits packets on a set of ports which are partitioned across multiple *line cards*.

2. A packet forwarded by the edge router belongs to a *packet flow*, where packet flows are composed of related packets.

3. The edge router handles upwards of 100,000 independent packet flows concurrently, as of 2005.

4. A packet received from a port is transferred to a block called the *ingress block* for processing, then moves off the line card through the *switch fabric*. The switch fabric routes the packet to its outbound line card. On the outbound line card, the packet is moved through the *egress block* for additional processing.

5. The supported line rate for an ingress block can be relatively high at 40Gb/s.

6. Packet order must be maintained within the router, or restored prior to scheduling the outgoing transmission, as the in-bound packet order is used when determining the out-bound packet order.

7. A core service provided by an edge router is the enforcement of service level agreements, as in the Differentiated Services application. This application on high-performance edge routers is typically composed of units of work or *tasks* which use large, randomly-accessed data structures of 100,000 or more objects stored in off-chip memory.

1.2.2 DiffServ: Our Edge Router Quality of Service Application

A fundamental part of the work performed by edge routers is ensuring quality of service. Differentiated Services, or *DiffServ*, is a widely-used application implementing QoS for edge routers by allocating traffic bandwidth to packet streams and monitoring them for compliance. DiffServ is composed of a chain of packet processing tasks executed for each packet, and includes some functions commonly used in many other packet processing applications. We use this core packet-processing application to serve as an important example of the execution performed on behalf of packets, rather than a benchmark suite of individually-executed packet-processing functions, for several reasons. Although several benchmark suites have been created, no dominant suite has emerged to provide universal basis for comparisons. Existing benchmarks are also frequently drawn from existing operating system code or designed for all-software routers and include heavy-weight administrative tasks, making them poorly designed for large router platforms. Lastly, a benchmark represents a single task while a packet will have multiple tasks executed on its behalf: measuring individual benchmarks overlooks the effects of communication, load balancing, and heterogenous execution resulting from more complex applications. Using DiffServ as our representative application sacrifices some breadth provided a benchmark suite, but it compensates by offering depth and applicability to our problem domain. We use multiple implementations in our experiments.

1.2.3 Flexibility Offered by Programmable Hardware

Processing packets on programmable hardware allows the system to be flexible. Programmable hardware has been suggested as a replacement for some hardwired packet-processing platforms to offer more flexibility, albeit at the possible cost of diminished processing speed. We can define two axes for measuring system flexibility. On one axis we can measure the *static flexibility*; the ability to change the implementation to adapt to new requirements. A completely hardwired system would be the least flexible in this respect while a system implemented entirely in software on programmable cores would be the most flexible; in the middle lies the hybrid solutions using programmable cores with hardware accelerators. Along the other axis is the *dynamic flexibility*, the ability of the system to support runtime variations for individual packets. The most flexible system would allow significant execution variations at runtime to meet the needs of individual packets; it would support a variety of functions or function ordering. A less flexible system would allow minor execution variations within individual functions. Dynamic flexibility is an important asset for programmable hardware, allowing it to manage a wide variety of packets without removing them from the general processing platform. Our target platform, outlined above, supports both static and dynamic flexibility.

1.3 Methodology

The studies we report in this dissertation use a model of the ingress block of the edge router line card executing hand-coded implementations of a portion of the DiffServ application, as described in the previous section. We analyze the performance of our system using a simulation-based methodology as well as analytical models.

We simulate the execution of our application on the chip multiprocessor platform using an instruction emulator written in-house to model our chip multiprocessor. We provide infinite remote memory bandwidth, which acts as a simplified version of our problem of using the compute hardware efficiently to consume the available memory bandwidth. The remote memory latency is set as an experimental parameter.

The input to a network architecture system is the network traffic. The packets can vary in their size and execution requirements as well as their relationships to each other. Some systems employ a very simple network traffic model: all packets are independent, they are all a fixed size, and each packet requires the same processing. These systems are useful for isolating specific parameters and restricting the number of variables. Other systems use traces of actual network traffic, or packets whose parameters may be statistically distributed over some range, to represent more realistic traffic behavior. Network traffic may also be *channelized* or *unchannelized*, referring to whether the total bandwidth of the network traffic is divided among several independent connections or all part of single connection. For example, the edge router line card we described in chapter 1 may get unchannelized traffic from a single port or channelized traffic from several ports. The network traffic we use in our experimental framework is composed of fixed-size packets requiring the same processing: all packets processed concurrently are independent of each other. We use this simplified problem to delineate the key application and system characteristics, and use those results to extrapolate to more complex execution.

Another variable in the network traffic composition is the required size and access patterns of the persistent data structures. The packets in our network traffic require a persistent data structure that contains hundreds of thousands or millions of elements. The data structure size forces the data to be stored off-chip, and the data is accessed randomly with very little temporal or spatial locality.

1.3.1 Measuring Application Performance

We report the performance of our application using three metrics. We measure the throughput reported as the *line rate* in bits per second. Two other performance metrics are *packet latency*, which measures the time the packet is resident on the router, and *packet jitter*, which is the variation in packet latency.

Packet latency is composed of *active* cycles, when a processor is executing instructions on behalf of the packet, and *inactive cycles*, when a processor waits for a resource such as data. Techniques for increasing throughput may have positive or negative second-order effects on the active and inactive cycle counts. Increasing a packet's inactive cycles may also degrade throughput as the packet store resource limits are reached; once the maximum number of packets are stored, no more packets may be accepted until a resident packet graduates. Finally, some network protocols are sensitive to packet latency, requiring those packets which take too long to reach their destination to be dropped.

Jitter can also be affected by optimizing throughput. Optimization can introduce jitter, or it can magnify inherent jitter by intertwining the execution of independent packets, as when multiple contexts are executed on a single processor. As with latency, some protocols are very sensitive to packet jitter.

1.4 Experimental Results

We discuss the results of two studies which describe techniques and performance implications of using processor resources efficiently. In the first study we evaluate methods for allocating work across an array of simple processor cores. In the second study we analyze several techniques for increasing throughput performance by loosening the instruction fetch bandwidth bottleneck.

1.4.1 Assigning Work to Processor Cores

Several proposals have been made for assigning work to an array of processor cores, but they have not focussed on our application domain or compared the costs and benefits of each method. In this study we present a comprehensive list of methods for using multiple processors to exploit task parallelism. We compare implementations of our application using a subset of the task parallel methods to a pure data parallel implementation, which relies solely on the parallelism of independent packets. We show that although the hardware contexts assigned to each processor core can be a performance bottleneck for our data parallel implementation, we can overcome the bottleneck by using *virtual contexts*, which virtualize the hardware contexts to allow each processor to support more independent threads than it has hardware contexts. We evaluate the performance effects of using virtual contexts and describe analytical models for the processor stall, packet latency, packet jitter and memory port traffic based on key system and application parameters.

1.4.2 Reducing the Instruction Fetch Bandwidth Bottleneck

Once we have mapped the application work to the processor array to minimize the total processor stall, the packet throughput is determined by the instruction fetch bandwidth. We evaluate several techniques for increasing the packet throughput, including compressing the executable into fewer instructions using extensions to the instruction set architecture tailored to characteristics of our application domain and using processor configurations which increase the instruction fetch bandwidth by offering dual instruction issue. We analyze the effects of our optimizations on processor resource utilization and contention and show that our optimized application executes with higher throughput and less processor contention, but copying the virtual contexts makes significantly more demands on the local memory ports. We also show that the dual issue configuration which issues two instructions in parallel from the same context, reducing the cycles required to execute the instructions for a single packet, increases bursty references to local memory while the configuration which issues one instruction from each of two independent contexts in parallel smooths the local memory reference frequency by staggering the context execution. Lastly, we observe that the branch frequency for our application is increased by performance optimizations which reduce the instructions executed per packet, providing more opportunity for pipeline stalls which could degrade the packet throughput. We evaluate a fine-grained scheduling algorithm to keep our instruction pipeline full by switching between executing contexts and analyze the resource utilization and contention using revised analytical models.

1.5 Conclusions

In this dissertation we report work using several implementations of a critical packet processing application executed on a concrete model of a programmable hardware platform. As a result of these studies we conclude we can leverage specific characteristics of our application domain to improve processor utilization and manage contention for resources. We evaluate methods for offering the benefits of exploiting task parallelism within the context of a more efficient data parallel implementation. We observe that processor hardware contexts are a critical resource to supporting data parallel implementations and propose virtualizing them to remove a significant performance bottleneck. By exploring the effects of this technique on resource utilization and execution performance we develop analytical performance models based on key system characteristics.

1.6 Road map

The rest of the dissertation is laid out as follows: we begin by discussing the related work in chapter 2. Chapter 3 reviews the experimental apparatus used in our studies, reported in chapters 4 and 5. We conclude and suggest future work in chapter 6.

Chapter 2

Related Work

As wide-area networking has burgeoned, both academia and industry have focused on developing and using programmable hardware to support it. A wide range of networking platforms exist, including

- a general-purpose x86 box which executes networking functions, along with a variety of other applications, under a commodity operating system,
- a router implemented in software, under a commodity OS, on a dedicated x86 box,
- a router implemented in software on one or more commodity network processor chips, or NPUs,
- a router implemented entirely in hardwired (non-programmable) hardware, and
- a router implemented using a mix of hardwired and programmable hardware.

Each platform may be suitable for several of the many available niches, which can include an individual's desktop machine, a small business gateway, or an Internet backbone router.

In this chapter we discuss prior work related to this dissertation. We begin by reviewing published academic research which relates to our work, including studies on mapping packet processing applications to parallel processor cores and using the resources of individual processors efficiently. Because of active concurrent development in industry, we also highlight some relevant commodity systems. We discuss some opportunities for further research, and close with a description of the work presented in this dissertation.

2.1 Academic Research

The computer architecture problems in networking have posed some infrastructure challenges to the academic community, as the commonly used tools and paradigms developed for the general-purpose or scientific computation research community may not fit. Consequently, academic research efforts have been divided between investigating design problems and developing the infrastructure needed to enable evaluations. Researchers have directly addressed a wide variety of design questions, including programming models, compiler optimizations, memory system designs, power issues, application algorithms, implementations and performance analysis, macro-architecture comparisons, and novel chip designs. Research infrastructure development has itself been divided between proposing methodologies and creating tools; these efforts have embraced network application benchmarking as well as simulation and performance modeling frameworks. Much of the academic research on the architecture of network processors has been collected into volumes 1, 2 and 3 of *Network Processor Design Issues and Practices* ([8], [9], [10]). These books contain the work drawn from this area's core forum, the Network Processor Workshop.

This dissertation is primarily concerned with using data parallelism to make the most efficient use of programmable hardware on a chip for processing packets. In this section we discuss work in the area of mapping network applications to parallel processors. The systems in the studies we report vary in their goals, implementations, and performance measurements. Some target applications included administrative as well as packet-processing tasks: the administrative tasks are performed periodically and typically include much more computation than packet-processing tasks do. Target packet-processing applications with multiple tasks may be structured as a chain of dependent tasks or they may include several independent tasks which can be executed in parallel. Although the prior work has a shared motivation of using programmable hardware to provide flexibility, the kind of flexibility which is supported varies: some proposed systems support only the ability to update or replace the software while others support run-time variations in packet processing. Some systems include data and/or instruction caches local to the processors; others do not cache data and store the complete executable for each processor in its local instruction buffer. The persistent data structures may be stored on- or off-chip. Authors may measure performance using individual task benchmarks or larger applications which link tasks together: their performance metrics have included packet throughput, packet latency, and processor utilization. They may compare the performance of configurations using the same number of processor cores, or they may assign different processor resources to specific configurations.

To use an array of processor cores we have to decompose the packet processing work onto parallel processors. A sequence of two papers, [22] and [15], addresses the problem of parallel decomposition using the same platform, albeit with two different goals. Both papers model members of the Intel's IXP family, the IXP1200 and IXP2400, which are designed to support applications performing frequent non-local memory references. Each processor maintains multiple hardware contexts, allowing it to switch between contexts and continue doing useful work after a context has stalled waiting for remote data.

The authors of [22] propose a methodology for mapping applications onto NPUs similar to the IXP2400. They provide a case study by partitioning their application, IPv4 forwarding plus DiffServ, to execute on the eight processors of an IXP2400. Their methodology sets budgets for instructions and cycles of memory latency stall per packet for individual processors, then partitions the application into blocks at the task boundaries and assigns the blocks to pipeline stages. A single stage may include one or more blocks: a stage which uses less than a processor's budget is assigned to a single processor, a stage which uses more than one processor's budget is divided up and assigned to multiple processors. The goal of the methodology is to use the hardware resources efficiently to maximize throughput using minimal application information, and it lays some groundwork for experimental analysis of this and other techniques for employing a processor array for handling packets.

In [15] the authors use a similar platform, the IXP1200, to address the question of how many pipeline stages should be used to implement an application. They partition the IPv4 application into one, two, four or eight load-balanced blocks and map each block to its own processor. Every implementation uses the same number of processors; the one-block version is copied onto eight processors, the two-block version is copied onto four processor pairs, etc. The network traffic is composed of packets of equal length; the packet size (and thus inter-arrival rate) is a configuration parameter. The network traffic is divided among sixteen channels. The performance metrics are the packet latency and processors utilization, the latter measuring the relative efficiency of the different mappings. The experiments are largely performed using an analytic performance model, whose demonstration is a main contribution of the paper. Based on the results, the authors conclude that multi-stage pipelines are inferior to single-stage implementations for two reasons: single-stage pipelines make better use of local memory for channelized traffic, and they avoid the communication between pipeline stages. The authors provide the experimental analysis measuring throughput for the partitioning method which results in equal-sized blocks for balanced processor loads, and identifies communication as a source of execution overhead. This work contributes to the groundwork begun in [22], discussed above, evaluating strategies for keeping a processor array executing a packet-processing application efficiently.

The authors of [6] present an optimizing compiler for Intel's IXP2400, which automatically assigns work to the parallel processor cores and the larger StrongARM processor according to per-packet instruction and memory stall budgets. The parallel processors could support up to 700 instructions and two off-chip DRAM references per packet. They presented the results of compiling three network applications, *L3-Switch*, *MPLS*, and *Firewall*. The authors report reduced throughput for multi-stage packet pipelines due to communication overhead: the processors communicated through off-chip DRAM, whose limited bandwidth acted as a performance bottleneck. The authors build on previous work to automate the partitioning of an application to a processor array: although they cite communication as the performance constraint on multi-stage packet pipelines, their conclusion applies to systems with inter-process communication through off-chip memory and restricted off-chip bandwidth.

Other researchers have also explored the question of pipeline length. The authors of each of two papers, [13] and the later paper [37], note the prevalence of processor pipelines in NPUs and present comparisons between varying numbers of processors using an abstract model. These papers each employ an abstract model of a chip multiprocessor and measure packet throughput. Unlike [15], both of these papers show that additional stages in a pipeline lead to increased performance. They base these results on experiments which compare execution times using increasing numbers of processors: a single stage pipeline is executed using one processor, a two-stage pipeline is executed using two processors, etc. These papers are primarily concerned with using a processor array to exploit intra-packet parallelism: their conclusions would best apply to a system with little packet independence in the network traffic, where the processing of dependent packets would be serialized and could starve some processors of useful work.

In addition to the static mapping approaches discussed above, dynamic approaches have also been explored: the authors of [30] present the programming environment and dynamic runtime manager NEPAL. NEPAL is designed to help a network processor provide both high performance and flexibility by simplifying application programming and extracting application-specific data to help dynamically schedule application tasks on a processor array: the authors conclude that it also significantly improves throughput compared to data parallel implementations. They present experimental results from trace-driven executions of the authors' benchmark suite NetBench, designed to represent a variety of packet processing and administrative tasks, on two modeled systems. The first is Intel's IXP1200, which places its processor array on a shared bus. The second system is the Cisco TOASTER, which uses neighbor-to-neighbor communication between processors. NEPAL partitions the application into modules to optimize for either a balanced processor load or minimized communication overhead. The authors use packet throughput as their performance metric, normalized to a data parallel implementation where each processor core executes a copy of the entire benchmark. In addition to the results using NEPAL, the authors present execution results using benchmark tasks which were statically scheduled by hand. The static- and dynamically scheduled pipelines both showed significant improvement over the data parallel execution in most cases for implementations compared executing on the same number of processors. As with the work in [6], the object is to automate the partitioning of an application across a processor array, and as with the work reported in [13] and [37] discussed above, the authors of this work exploit the thread-level parallelism available in the processing of a single packet. Their conclusions, which contradict those in [6] and [15], are based on the performance effects of the instruction and data cache miss rates, which appear to overwhelm any communication overhead incurred, while the intra-packet parallelism is exploited to reduce the packet latency rather than increase packet throughput as in [13] and [37]. These results imply that the conclusions presented here apply to systems with enough packet independence in the network traffic to keep the processor array occupied, employ instruction or data caches but execute with a high miss rate, and execute enough computation in each partition to make the communication overhead negligible by comparison.

Researchers have gone beyond the question of how to map applications to parallel processors to ask whether a different type of parallel processing might be more suitable for packet processing applications. The authors of [32] propose using a CMP vector processor to perform Single Instruction, Multiple Data (SIMD) ([12]) execution of packet processing applications. They stress the static flexibility of evolving applications as the primary motivation for using programmable hardware and propose using a vector processor to gain both performance advantages over independently executing processor cores and flexibility advantages over hardware accelerators. The study presents a vector processor design. The experiments are executed on a simulated model using three benchmarks, which allow some execution path variations for individual packets. The authors collected packet traces on a router in their own lab and interleaved them to create their experimental network traffic. They show the feasibility of implementing their benchmarks for a vector processor and analyze the data cache use, laying the groundwork for comparative analysis with the approaches outlined in the studies discussed above. Their focus on data caches implies that their conclusions are relevant for systems with network traffic which generate accesses exhibiting locality.

In order to optimize application execution we need to understand how to employ the resources of an individual processor. Several papers address this question in the context of network applications. The authors in [7] sought to determine how to distribute and schedule a fixed number of processor functional units while the authors of [31] propose using a streaming vector processor, which provides a programming model that allows the individual functional units to be scheduled in parallel.

In [7], the authors attempt to determine the optimal way to distribute a fixed number of hardware resources. They perform direct comparisons between four architectures: an aggressive super-scalar core (SS), a fine grained multi-threaded processor, which is SS plus multi-threading with zero switch penalty, a chip multiprocessor with an array of single-issue processor cores, and a simultaneous multi-threading (*SMT*) processor. Their experiments use three benchmarks, each executed in isolation and in conjunction with an operating system, with no runtime variation in packet processing execution specified. Each architecture was configured with comparable issue widths, and the experimental results were reported in packets per second. This work treats network processors as alternatives to all-software x86 platforms. The authors found that the benchmark and operating system execution were both processor-bound, which gave no advantage to the traditional multi-threaded processor. They further concluded that the SMT core was the only architecture to perform well under both the single-threaded execution of the operating system and the highly parallel execution of the packet processing benchmarks. Although this relatively early work includes a CMP architecture, the focus is on aggressive super-scalar uniprocessors; the software, which is compute-bound and requires a single-threaded operating system, is targeted to an all-software router. The authors of [31] build directly on [32] and propose using an Imagine-like architecture ([11]), with multiple processor cores composed of parallel functional units, to execute specific network processing functions. The authors conclude stream-based architectures, which allow the processor resources to be scheduled in parallel using microcoded instruction, could be useful for network applications. Their experiments use two individual benchmarks executed on a model of the Imagine stream processor with fixed size packets (the size set as a parameter) and with network traffic from a collected trace. The reported performance metrics are packet latency, functional unit occupancy, and the percent of total execution time spent in the kernel, as opposed to performing stream operations or stalling.

Two papers discuss the instruction level parallelism available in network applications. In [25] the author reports on processor utilization executing an abstract application using traces of actual network traffic collected on low-throughput links. The author concluded that limited packet independence suggests systems should exploit intra-packet parallelism and instruction level parallelism to improve throughput. The authors of [23] introduce a suite of network processor benchmarks and include their analysis of the available ILP.

2.2 Commodity Programmable Hardware

Companies have been supplying commodity network processors to provide specialized programmable hardware platforms for routers for several years. Because this area has been developing concurrently in industry and academia, looking at existing commodity hardware shows what implementations have been available and successful in the marketplace while the research has taken place. In this section, we will give an idea of the options available and the shifting nature of the market. Previous work has surveyed, classified, and discussed commodity network processors ([33], [8], [9]): we will review a few examples of commodity NPUs from the two largest vendors and one much smaller supplier.

Although NPUs were originally positioned to compete with the general-purpose x86-based platforms, the x86 remains widely used and some NPU suppliers have been changing their focus to re-target their products to support higher throughputs (10 Gb/s or more) and compete with hardwired solutions. Commodity NPUs have a small but growing market ([1]), and, although the field of suppliers has started to narrow, there are still many companies competing with a variety of chip designs. No single supplier or design has grown dominant; an NPU may or may not have special purpose hardware accelerators in addition to many simple parallel processor cores, a few large, fast processor cores, or some combination of the two. The programmable cores offer an additional dimension; they can support very general execution or be specialized for network applications, executing as independent cores or coordinate for SIMD execution. The breadth of options and still-lively competition indicates that there has been no consensus reached on the best way to employ programmable hardware for network applications. We discuss three different platforms below.

Intel produces the IXP family of network processors @@@ cite, which are not only widely used as research platforms but also had 21% of the market share in 2004 (up from 15% in 2003) ([1]). The IXP2800, which is targeted to support 10 Gb/s half-duplex throughput, has a single StrongARM processor as well as an array of sixteen simpler parallel processors which share several hardware accelerators. Each processor supports multiple hardware contexts to maintain independent instruction streams. A processor can execute instructions from its local control store; any changes to the local stored instructions require the simple processor to stop executing until the changes are complete. Packet processing is meant to occur on the array of simple processors, which have hardware support for register-to-register communication between fixed pairs of processors in a line (processor A can exchange information with processor B, which can also communicate with processor C, etc.). The packet processing functions can be partitioned across the processor array, which then acts as a packet pipeline. In theory, the IXP can also support a more data parallel implementation by assigning each packet to a single processor; in practice studies (including those published by Intel) based on an earlier version of the architecture, the IXP1200, almost never employ this mode.

AMCC, which had 31% of the market share in 2004 (down from 37% in 2003) ([1]) produces the nP7510 network processor, a member of the nP family, targeted to support 10 Gb/s half-duplex throughput. The nP7510 contains six processor cores which support hardware multi-threading, aided by on-chip hardware accelerators. Each nPCore is designed to process packets to completion, as we advocate in this dissertation; unlike the Intel products, the nP family does not provide hardware support for employing processors cores in a packet pipeline.

EZchip ([2]) had a small slice of the NPU market, around 5%, but its revenue tripled between 2003 and 2004 ([1]). The EZchip NP-2 network processor, which is intended to deliver 10 Gb/s full duplex throughput, uses Task Optimized Processing

(TOP)cores. Each TOP is optimized for one of four packet processing tasks; different kinds of TOPs are arranged into a packet pipeline. A TOP is not multi-threaded, but multiple TOPs at the same pipeline stage can process multiple packets.

We present these three vendors because they are each a note-worthy presence in the NPU market: Intel and AMCC earned the two largest percentages of market share, and EZchip, although a much smaller player, has grown very rapidly and is focused on supporting high throughput. As a group, their products populate only a portion of the design space: each design uses multiple processor cores on a chip for packet processing, and the processor cores are executed in MIMD fashion. The designs differ along several axes. Where Intel builds in support for considerable variability managing the hardware resources, both AMCC and EZchip set more rigid constraints but differ in how the processor cores are used. These design variations could imply that the market has not settled on a solution for efficient packet processing. Of the three, the AMCC products are the best example of the design principals we advocate in this work.

Companies offering large routers are beginning to incorporate programmable hardware into largely hardwired designs. The Cisco Carrier Routing System (CRS1) @@ cite, a large router with programmable hardware uses hardware and software architectures to provide speed and flexibility for millions of packet flows. It uses the Silicon Packet Processor programmable ASIC and offers the choice between flexibility and performance, giving the router access to more positions in the wide area network. Juniper Networks, Cisco's competition, offers the T640 with programmable ASICs. Both companies chose to develop their programmable hardware in-house rather than choose off-the-shelf parts. These large routers are examples of the target platform used in this dissertation: high performance routers which include customized programmable hardware to perform work which requires flexibility. The purpose of this dissertation is to identify efficient techniques for using programmable hardware to allow these large routers to offer more services which require flexibility to higher volumes of traffic.

2.3 Unexplored Territory

Previous research in executing network applications on programmable hardware has largely focused on lower-end systems. Some systems execute applications which include the longer-running administrative tasks as well as the tasks for processing individual packets. Persistent data structures are small enough to be stored on-chip or exhibit enough data locality to benefit from caching, implying network traffic with a relatively small number of independent packet streams. These studies have focused primarily on measuring packet throughput performance without reporting packet latency and jitter, which are affected by scheduling algorithms and contention for resources. Commodity systems started as alternatives to the x86, acting as platforms for small all-software routers. NPU's have recently begun targeting higher packet throughputs and a few large routers have recently included programmable packet-processing hardware, but information about vendors and products does not give much understanding of the reasons for any successes or failures: it does not tell us what technical advantages the individual designs have, nor how they are being used by customers. In particular, we don't see how platforms are used in practice: how they are positioned in the network, the functions they are executing, and what throughput they are supporting. Many prior studies have not emphasized systems which support dynamic flexibility, the ability to adapt packet processing to the needs of individual packets. Overall, the work in this area does not give a coherent picture of the characteristics and performance of large router packet processing, which requires dynamic flexibility and high packet throughput with constrained latency and jitter, large randomly-accessed data structures, and short tasks.

Studies which evaluate parallel decompositions onto multiple processor cores usually consider only one or two methods of partitioning applications, such as comparing a data parallel implementation against a balanced packet pipeline: there has been no comprehensive analysis of partitioning techniques to evaluate their benefits and associated costs. While previous studies have reported the performance effects of specific configurations, they usually do not isolate the effects of individual sources of performance benefits and overheads resulting from the configuration executing on the particular system, such as the effects of instruction and data caches. Although the results of each work are internally consistent, they do not support extrapolation to systems with different parameters and sometimes appear to conflict with results from other research. The studies which report the efficiency of data vs. task parallel implementations stop short of comprehensive analysis of the advantages and limitations of each technique. As a result, existing studies make or assume opposite conclusions without clarifying the limitations on their scope.

Lastly, research on the performance of packet-processing applications has not focused on the performance effects of instruction-level executable characteristics. The instructions implementing applications and benchmarks are largely unexamined: studies which budget the number of instructions executed per packet treat them as a single undifferentiated block. Although packet-processing tasks are short sections of code repeated for many or all packets, there has been little analysis of their characteristics and how they may be used to optimize the executable instructions.

For this dissertation our target system is a large edge router executing a short packet-processing application, composed a chain of dependent tasks, on programmable hardware. Our application's persistant data structures are large and randomly-accessed, requiring us to store them in off-chip DRAM memory.

The first part of our work explores parallel decompositions for mapping our network application onto parallel processor cores. Exploiting task parallelism has been a widely-used technique in both industry and academia, and it deserves a rigorous evaluation: we build on the previous analysis of mapping techniques and present an experimental analysis of the costs and benefits associated with a comprehensive set of methods for statically mapping our application to a MIMD processor array. We use only inter-packet parallelism as our application has no task parallelism which can be exploited for a single packet. We do not use data caches as our data objects exhibit little locality, and we use a perfect instruction cache to allow us to de-couple the mapping technique performance from the question of managing the local instruction buffer. For our purposes we would like to determine how to use the available hardware efficiently, so we compare executions which each use the same number of processors. We identify the most promising technique, which exploits data parallelism rather than task parallelism, as well as a potential resource bottleneck at the number of hardware contexts available to a processor. We propose virtualizing the hardware contexts to improve multi-threading support and offer performance models to predict the effects of virtual contexts on resource use and performance.

In the second part of this dissertation we evaluate several ways to loosen the instruction fetch bandwidth bottleneck and improve packet throughput. We identify some typical characteristics of packet processing tasks and leverage them by extending our instruction set architecture to optimize the instructions executed per packet. We also show the performance effects of exploiting instruction-level parallelism in a single context, and issuing in-order instructions from multiple virtual contexts. We analyze the effects of our optimizations executed using virtual contexts, measuring throughput, contention and resource bottlenecks, and branch frequency.

Although our analysis is performed on a simplified system using very uniform network traffic, where all packets require the same processing, our ultimate goal is a system which supports runtime processing variations: we review our results with that goal in mind and apply our findings to more complex execution.

Chapter 3

Methodology

The research we report in this dissertation is based on experimental results gathered by modeling the execution of a packet processing application on a chip multiprocessor or *CMP*. The CMP represents the ingress block on an edge router line card and the application executes the ingress portion of the Differentiated Services quality of service packet processing. In this chapter we discuss the experimental framework we use to generate the results we report.

Our experiments are designed to analyze a well-defined packet-processing system in our target domain to determine the parameters which are important to its performance. By closely examining a single concrete system we can develop intuition about the its behavior and provide a strong basis for comparison to predict the performance of other systems not analyzed here. We have constructed our experimental apparatus to isolate the performance effects of individual resources: to that end we largely focus on varying the availability of a single resource while allowing unrestricted access to the rest. We have also kept the processor modelling simple to focus on macro-architectural behavior.

We review our hardware platform in section 3.1, our software application in section 3.2, and the execution variation and network traffic in section 3.3. We discuss our simulation infrastructure in section 3.4. We profile our performance metrics in section 3.5.

3.1 Ingress Block Hardware

We use a model of the Stanford Smart Memories chip multiprocessor [26], a modular, reconfigurable architecture designed to serve several application domains, as a model for our edge router ingress block. The Smart Memories CMP shares characteristics with several network processor architectures, although it is not specifically designed to execute network applications: it has many simple processor cores on a single chip connected by a high-bandwidth network. The processors on the Smart Memories chip support multiple configurations: this design allows us to construct implementations with differing processor requirements and compare them directly on the same hardware platform.

The chip is composed of an array of processing tiles, each holding 128KB of local memory, a local interconnect, and a reconfigurable processor core with eight hardware contexts. Tiles are networked together by a local interconnection network, in groups of four, into a *quad* and each quad is connected to a global interconnection network. The processor core on a tile can be configured in several different ways. Our default configuration is to partition the core into two smaller processors, each using four hardware contexts and supporting single-issue 32-bit RISC-like instructions. The Smart Memories design also supports a dual-issue mode for each of the two processors on a tile: in chapter 5 we use this configuration to support our *Static_ILP* and *Static_SMT* executions. Lastly, the Smart Memories processor cores also support a streaming mode with a 256 bit instruction format, which combines all the processor resources on a tile into one processor and allows operations to be issued to each functional unit in parallel: we use this configuration to execute the *CPU_Config* implementation in chapter 4. The combined computation resources include two integer clusters each consisting of an ALU, register file and load/store unit and one floating point cluster with two floating point adders, a floating point multiplier, and a floating point divide/square root unit. The floating point units can perform the corresponding integer operations. Our experiments use between two and twelve processor cores, resident on between one and six tiles.

3.2 Ingress Block Software

We have chosen to perform our experiments using implementations of a simple, widely-used application made up of a chain of dependent packet processing functions or *tasks*, where a task is a unit of work associated with a data structure. The application represents the portion of Differentiated Services or *DiffServ* that would execute on the ingress block of our target edge router. The tasks we include are packet parsing, 5-dimension flow classification, and metering; the first two are fundamental to packet processing and shared by many applications. We chose these tasks because in a more fully developed workload they would provide a high degree of flexibility and are consequently more likely to require a programmable platform. The implementations we use are restricted to basic forms of the tasks to allow us to isolate performance effects. Our results will help us understand how to extend the applications to provide the additional flexibility they require.

Each implementation of the workload is hand-coded in MIPS assembly, which uses simple RISC-like instructions. The *parser*

1. verifies the IP version number,
2. compare the packet length to the minimum allowed length,
3. checks for options encoded in the header,
4. maps the protocol to the appropriate action through a jump table, and
5. extracts the relevant data from the packet header to prepare for the 5-dimension classification (the action dictated by the protocols in our packet stream).

The *classifier* matches packets to policy objects using a simplified trie modeled on a trie constructed by the Hi_Cuts algorithm [18]. Each internal node has eight children. Each leaf node represents eight policy objects. The copies of the root of the trie are stored locally at each processor; all other trie nodes are stored off-chip. Every packet follows a path through the trie composed of the root, a single internal node, and a leaf node. The purpose of using this simple and predictable trie is to show the processing and memory access requirements for each trie node while eliminating the variability of a deeper, more complex trie. This regularity allows us to isolate the performance effects of optimizations we employ. For each internal node in the path, the classifier

1. extracts the dimension type used to match a packet to a child node,
2. compares the dimension values of the packet and node, and
3. fetches the next node (on a match) or moves to information for the next child node.

For the leaf node, the classifier

1. extracts the dimension type used to match a packet to an offset in a jump table,
2. compares the dimension values of the packet and node,
3. uses the jump table offset to branch to the appropriate action code (on a match) or moves to information for the next offset, and
4. increments the appropriate statistics counters and fetches the matching policy, from a database of approximately 40,000 rules stored entirely in off-chip DRAM, to prepare for metering the packet.

The *metering* function performs part of the Quality of Service packet management. It implements the single-rate, three-color algorithm outlined in RFC3697 [19], reading from and writing to the policy object fetched by the classifier. To keep packet handling execution uniform, the system is constrained to prevent policy object access conflicts. No two packets will match to the same object at the same time. For each packet, the meter function

1. checks the flow rate logged in the policy object,
2. marks the packet,
3. updates the local copy of the policy object, and

4. writes the policy object copy back to off-chip storage.

The execution of the metering task changes slightly according to the state of the policy object, which causes the dynamic instruction count per packet to vary by up to five instructions.

Performance analysis of general-purpose computer architecture is traditionally done using a suite of benchmarks which are executed independently. The suite is designed to cover a range of representative application behaviors, but no single benchmark is necessary for realistic general-purpose architecture workloads. Although benchmarks are commonly used for network processor architecture studies, as of 2005 no single set has been standardized.

Benchmarks have some limitations in network architecture experiments. Unlike general computer architecture benchmarks, which tend to represent stand-alone applications, network processor benchmarks represent individual functions. In a real system a single packet will have several functions executed on its behalf and as a result the processor array will be executing a heterogeneous set of packet processing tasks, passing data between them. The performance effects of individual benchmarks will also vary according to their relative weights: this kind of execution behavior creates relationships between benchmarks which are not revealed when they are executed individually. Packet processing systems are also more specialized than general-purpose systems and include tasks, such as classification which are fundamental to most processing tasks in a way that individual general computer architecture benchmarks are not.

Establishing a truly representative application or benchmark suite is beyond the scope of this work. Rather, we give up the breadth offered by a set of individual benchmarks and instead focus on detailed analysis of multiple implementations of our single, widely-used application made up of several dependent tasks shared by many packet-processing applications. This application forms a part of the concrete system we use as the platform for our experiments.

We also model the extremely simple applications whose execution consists of a fixed number of instructions and remote memory requests. These applications perform no computation and do not modify any persistent data: their sole purpose is to clearly demonstrate a specific performance behavior based on the number of instructions and references.

3.3 Other System Characteristics

Our ingress block workload has static flexibility by virtue of being encoded in software. Our target system has dynamic flexibility and therefore could support significant variation in the number and ordering of tasks in the application. However, the application we execute for our experiments includes only minor execution variations within individual packet processing tasks; each packet in our network traffic is processed with the same tasks in the same order. We make this simplification in order to clearly isolate and understand the performance effects of specific system characteristics, and we use our performance models to predict the behavior of systems with more variability.

Our DiffServ application is executed on network traffic that is unchannelized, meaning that individual packets are accepted into the ingress block chip in series rather than transferring several packets in parallel on separate channels. The network traffic is composed of equal-length packets of forty bytes each. This packet size represents the minimum size packet for our target system; since our application performs the same amount of work per packet regardless of the packet size, making each packet small forces the system to perform at the highest required rate. All packets processed concurrently are guaranteed to be independent unless specified otherwise, modeling the common case for our edge router supporting a large number of independent packet flows.

3.4 Modeling Execution

Our event-driven simulator models the Smart Memories CMP by emulating multiple processors, each with multiple hardware contexts.

We use a simple model of the pipeline, executing each instruction in a single cycle. The cycles of pipeline stall inserted after each branch is set as simulator parameter:

in most cases we do not introduce branch-related pipeline stalls. Branch execution can be managed to prevent some or all pipeline stalls in a number of ways, and for the purposes of many of our evaluations we would like to isolate the performance differences between application implementations from the effects of branch behavior. When we analyze branch behavior we introduce a two-cycle pipeline stall after each conditional branch. A single register load takes two cycles, managed by exposing a load delay slot in the executable.

The processors on our CMP include multiple hardware contexts which store the state of independent threads. Switching between contexts takes zero cycles and incurs no performance penalty. The processor also supports virtualized hardware contexts, which allow the processor to support more threads than it has hardware contexts. The state of idle contexts can be buffered in local memory: the number of registers required to hold the context state is set as a simulator parameter. Copying a virtual context between local memory and a hardware context can occur while the processor is executing instructions from some other thread stored in another hardware context. The virtual context copy occurs one register at a time, negotiating for access to a local memory port each time. Contexts are automatically stored to memory on a remote memory access if the processor is assigned more virtual contexts than hardware contexts. They are loaded back into hardware in round-robin fashion.

We can model the local memory ports in two ways. First, our system can execute with unlimited local memory bandwidth in order to isolate the performance effects of other resources: we collect the frequency of local memory requests. Second, we can simulate our system with a fixed number of read/write memory ports, set as a simulator parameter, to report the performance effects of limited local memory bandwidth and potentially bursty memory requests. When the simulator models a fixed number of read and write ports they are scheduled as follows:

1. The executing context is given the opportunity to schedule a block of between one and eight contiguous cycles required by the next instruction. If the memory port is busy, the instruction is not executed and another attempt is made on the next cycle the context is scheduled to execute.

2. If more than one context may issue instructions during a cycle, the contexts attempt to schedule the ports in a fixed order.

3. After the executing contexts have had the opportunity to schedule a local memory port, a context being copied out to local memory may request a single write on the write port and a context being copied in may request a single read on the read port.

The memory port scheduling algorithm ensures that an executing context will conflict only with another executing context, not with a virtual context copy.

The local memory is software managed. Data is located either in the local memory or in off-chip DRAM; in the latter case, the remote data objects are copied into and out of local memory. We do not model read and write conflicts in local memory. The remote memory latency, the time to complete a read or write access to off-chip memory, is set as a simulator parameter and does not change due to memory or network contention. Our system does not support data caches because the persistant data used by our application has little locality and the cache miss rate would be quite high: the processor would still need to manage long latencies, and our round-robin scheduling would keep contexts waiting for access to the processor even after a cache hit made the data immediately available. An on-chip data cache could help free off-chip memory bandwidth, but that is outside scope of this dissertation. Instructions are always stored in local memory, allowing us to isolate the performance effects of other parameters. Instruction buffer management is an important topic for this application domain, but it lies outside the scope of this work.

3.5 Performance Measurement

We measure throughput, packet latency and jitter, and processor utilization for each of our experiments. All measurements are based on processing 24,000 packets. Packet latency and jitter numbers are collected after the first 4096 packets have been processed. Throughput is calculated from the number of elapsed cycles and the number of packets graduated, based on a 1 GHz processor frequency. Packet latency is calculated as the elapsed cycles between the packet's entrance in the initial work queue and packet graduation; the median packet latency is reported. Packet jitter is calculated as the difference between the shortest and longest packet latency values.

Packet throughput is the prime performance metric for packet-processing systems: if the system cannot process packets at the same rate it receives them, it will have to drop packets from the network traffic. We track packet latency and jitter because they are important system metrics which are affected by many of the same parameters which determine throughput. Evaluating system configurations or optimizations requires that we measure their effects on all three metrics.

Chapter 4

Scheduling Multiple Processor Cores

Network traffic with independent packets offers parallelism which can be exploited by executing work on multiple processors.

An application can be scheduled across multiple processor cores to exploit

1. data parallelism, by placing a copy of the entire application on each processor and distributing packets across the processors,

2. task parallelism, by placing a portion of the application on each processor and requiring packets to transfer between processors before they are graduated, or

3. a combination of data and task parallelism.

A data parallel implementation may distribute incoming packets to an array of processors, but it processes a single packet entirely on one processor. A packet-processing application in our domain generally requires a relatively small amount of computation which uses multiple remote data objects. A processor handling one packet at a time may spend most of its cycles idle while it waits for remote data accesses to complete: the processor may be used more efficiently by processing multiple packets concurrently, which can provide the processor with additional work to perform. Each packet is assigned to an independent context: the processor performs a context switch after a remote memory access and continues executing from another context. Although a data parallel implementation can process multiple packets in parallel, the tasks for each packet are executed in series.

A task parallel implementation uses an array of processors as a packet pipeline. The application is divided into stages and each stage is executed on a separate processor. All packets travel down the pipeline, transferring between processors, until the processing is complete. A processor in the packet pipeline may handle multiple packets concurrently as described above if the stage it executes includes an embedded remote memory reference. If there is no remote memory reference or it occurs at the end of the stage, the processor can execute the stage completely for one packet before beginning the next.

Partitioning an application to exploit task parallelism can offer one or more advantages for processing packets, depending on how the application has been partitioned. Those benefits may include:

- preventing contexts from becoming a performance bottleneck by executing from a single context on each processor,
- speeding up execution of some pipeline stages,
- reducing packet latency, and
- reducing demand on local instruction buffers

The processor's ability to switch between packets and prevent idle cycles will be limited by the number of contexts it can support. A pipeline stage which has a remote memory access only at the end of the stage allows the processor to handle each packet without waiting for data. The packet is transferred to another processor immediately after making the remote memory request, and the next processor does not begin handling the packet until the memory access has completed. Since the pipeline stage is not interrupted by long memory latencies, the processor can execute the stage completely for each packet before starting the next packet, all from a single context. It can continue executing as long as it has work, i.e. a packet with a local copy of the data object(s) it needs for the stage. Every processor in the packet pipeline can execute from a single context if every stage includes remote memory requests only at the end: the application uses the remote memory accesses as the boundaries of the partitions.

Partitioning the application across multiple processors also offers the ability to use heterogeneous processor configurations. The configuration of each processor can be optimized for characteristics of its stage and some stages can be executed faster than with a default configuration. The partition boundaries must be set according to the application characteristics.

If an application has independent tasks, those tasks can be assigned to separate stages and executed in parallel for the same packet. Dependent tasks for the same packet must be executed in series. Executing multiple stages in parallel for a single packet will reduce the packet's latency but it will not increase total packet throughput: the total number of processor cycles required to handle the packet remains the same.

A data parallel implementation, which handles a packet entirely on a single processor, requires that the entire application executable be stored or cached in every processor's local instruction buffer. Task parallel implementations assign separate executables to each processor, implementing only a portion of the application. The demand on the local instruction buffer is less than the full application: this may reduce instruction fetch latencies and instruction transfers for processors using instruction caches, or allow larger applications for processors not using instruction caches.

Partitioning an application across several processors introduces two sources of overhead to packet handling. Transferring the packet and associated state between processors adds communication overhead: when an application is partitioned, all the live state must be transferred across the partition boundary between processors. A packet processing task typically has three kinds of live data: the packet information, one or more persistent meta-data objects, and any additional state, such as calculated values, required to continue executing. An application can be partitioned to minimize the additional communication overhead by dividing the application where the least amount of state must be communicated. Generally, a partition boundary just after a remote memory access reduces the live state to just the packet information, assuming the remote data object can be requested by one processor on behalf of another. The upper bound on context state includes the packet information, the meta-data, and all the register state.

Partitioning an application can also create processor load imbalance. When partitions are not equally weighted, the processors executing any but the largest partition will stall, either because their input queue is empty or their output queue is full. When individual processors stall the total throughput performance declines.

The amount of each kind of overhead added to the application is determined by the position of the partition boundaries. They may be set at locations in the application with little live data, to minimize the communication overhead, or to create partitions of equal weight and eliminate load imbalance. In most cases the partition boundaries cannot be set to minimize both sources of overhead simultaneously. If an application is partitioned to achieve a benefit listed above, it may not minimize either source of overhead.

In this chapter we will report the results of executing several implementations of our application which represent the comprehensive set of options for mapping our application onto an array of processors. We compare their throughput results and show that the benefits offered by exploiting task parallelism are linked to overhead which can significantly degrade packet throughput. Data parallel implementations avoid the overhead and use processor resources very efficiently, maximizing the packet throughput: however, they require multi-threaded execution to achieve their potential throughput. We propose reducing the performance bottleneck for multi-threaded execution by virtualizing the hardware contexts and model the processor utilization and contention for a varying number of virtual contexts executing our data parallel implementation. Lastly, we use our performance models to predict the performance of a packet processing application with significant runtime variation between packets in order to evaluate the performance for an application with greater dynamic flexibility. In the next section we review the experimental apparatus we use for this study. We report on the results of our experiments in Section 4.2 and discuss further conclusions in Section 4.4.

4.1 Methodology

In our experiments we execute five implementations of our application:

- Data_Parallel,
- MemRef,
- LoadBal_Min,
- LoadBal_Max, and
- CPU_Config.

In the *Data_Parallel* implementation a packet is handled by a single processor, which executes the entire application on its behalf. Each processor maintains four independent contexts which are stored in hardware. Instructions are executed from one context until it incurs a remote memory access and waits for data. The executing context is then suspended and another context resumes running. When all contexts are waiting for data the processor stalls.

The remaining four implementations exploit task parallelism by partitioning the application into stages, each mapped to a separate processor. The Smart Memories model used for these experiments offers simple software-managed queues for communication between pipeline stages; the queues are configured to connect two individual tiles and the queue elements are stored in the local memory of the receiving tile. Queue operations are performed with a single instruction.

The *MemRef* implementation partitions the application at remote memory accesses. Each of four processors executes a portion of the application; on a remote request a packet is transferred to the input queue of the next processor to await the data. Each processor executes a from a single context. The first stage includes parsing the packet and processing the root trie node using a permanent local copy. It fetches the matching child node, located off-chip, on behalf of the processor executing the second stage. Stage 2 processes the internal trie node and fetches the remote leaf node for stage 3, which processes it and fetches the policy object for stage 4.

Stage 4 performs the metering function and graduates the packet. As packets are transferred at the boundaries between processing tasks, no register state is live and only the packet meta-data object is copied between processors while the remote data object is fetched into the local memory of the target processor. Since the source and latency of the packet meta-data and remote data object are not the same, the objects are pushed onto two independent queues. This implementation avoids the hardware context performance bottleneck, since each processor uses only one context.

LoadBal_Min implements the application partitioned over four processors. This implementation minimizes communication and processor imbalance simultaneously to establish the lower bound on total overhead. Each partition executes the same number of instructions to balance the application across the processors and eliminate load imbalance overhead. LoadBal_Min also includes only a minimum of communication overhead; no register state is considered live, so only the packet meta-data and the task's current meta-data object (e.g. a trie node) are transferred to the next processor. The two data structures are stored together in local memory, and when a packet is transferred to another processor, they are treated as a single data object and pushed together onto the same receiving queue; moving the two data objects together reduces the communication overhead even below that of MemRef. Since some LoadBal_Min partitions continue to process a packet after it incurs a remote memory access, those partitions are executed on processors which use four independent contexts to keep the processor supplied with work.

LoadBal_Min is constructed to represent the lower bound on both sources of overhead: communication and processor load imbalance. This combination of lower bounds is not possible for our application because it has live register state which must be transferred between the balanced pipeline stages, which adds to the communication overhead. For this reason we have implemented LoadBal_Min as an *artificial* implementation, which mimics the execution of our application by fetching packets and executing the correct number of instructions and remote memory references, but does not perform calculations or change any permanent state. Executing this artificial implementation allows us to gather throughput performance results and compare them directly against our other implementations.

The *LoadBal_Max* implementation represent the upper bound on communication overhead and the lower bound on processor load imbalance. As with LoadBal_Min, each of its four partitions executes the same number of instructions and transfers the packet meta-data and task data object between partitions. Unlike LoadBal_Min, however, LoadBal_Max also considers all registers as live and includes additional instructions to copy the data between processors; this represents the upper bound on communication overhead combined with the lower bound on load imbalance. Similar to LoadBal_Min, LoadBal_Max is an artificial implementation designed to generate throughput performance results: it fetches packets and executes the correct number of instructions (including the register reads and writes to transfer "live" state) and remote memory references.

The *CPU_Config* implementation is an example of a task parallel implementation which partitions the application for execution on heterogeneous processors. CPU_Config uses two processor configurations. The default *integer programming* configuration, used by our other implementations, executes one MIPS instruction per cycle. It supports conditional branches and accesses to data stored remotely. The second, the *stream programming* configuration, uses microcoded instructions to schedule functional units in parallel but supports only loop-based conditional branches and accesses to local data. By microcoding a partition, the work encoded in several MIPS instructions can be expressed in a single microcoded instruction: those partitions executed using the stream programming model take fewer processor cycles to process a packet. Configuring the processors heterogeneously allows some computation to be compressed into fewer processor cycles using the stream programming model while other parts of the application, which cannot be expressed in the stream programming model, are executed using the integer programming model configuration. However, any performance improvement must overcome the overhead incurred by partitioning the application; individual partitions may execute much more efficiently, but the overall benefit can be limited by non-optimized partitions and can be easily overwhelmed by communication and processor load imbalance. As with the technique used in MemRef, this technique is relatively inflexible; partition boundaries are set according to the execution behavior of the code and the limitations of the stream programming model, and the number of partitions (and therefore processors) is fixed. The first stage, implemented using the integer programming model, parses the packet. The second, third and fourth stages process the trie nodes using the streaming programming model. Stage 5, the second integer partition, executes the classification action and stage 6 performs the metering function. Each streaming partition executes on the processing elements of an entire tile, or both processor cores combined, and processes two packets together. Each integer partition executes on a single processor and processes a single packet at a time. To supply the stream partitions with enough packets, two copies of each integer partition execute in parallel. Since each remote data access occurs at a partition boundary, the CPU_Config implementation, like MemRef, uses queues rather than multiple contexts to mask memory latency. Because our simulator does not currently support microcoded instructions, CPU_Config is an artificial implementation like LoadBal_Min and LoadBal_Max. Each pipeline stage executes a fixed number of instructions per packet: the number of instructions for the stream partitions was calculated using the Imagine [11] auto-scheduler adapted for the Smart Memories processor.

4.2 Experimental Results

Maximum Throughput When each of the implementations is executed at its maximum throughput the Data_Parallel performance exceeds that of all task-parallel implementations. Figure 4.1 shows the maximum throughput of each implementation divided by the number of processors required to execute it. Data_Parallel,

Figure 4.1: Maximum Throughput

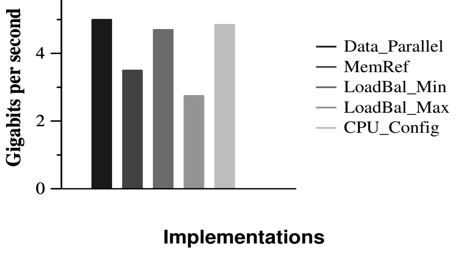
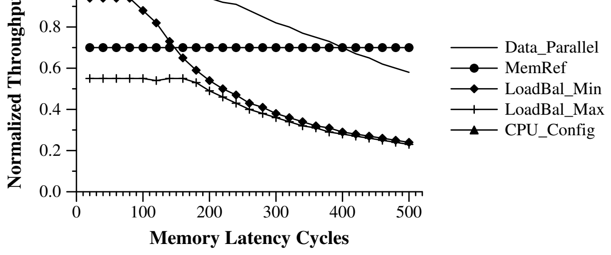


Figure 4.2: Normalized Throughput



LoadBal_Min, and LoadBal_Max execute four independent contexts, maintained in hardware contexts, per processor while MemRef and CPU_Config each execute a single context per processor. Data_Parallel executes with 5.0 Gb
s. LoadBal_Min, with a balanced processor load and the communication lower bound, supports 4.7 Gb
s while LoadBal_Max, which represents the upper bound on communication, can support only 2.75 Gb
s. MemRef, which uses minimal communication, has significant throughput penalty due to processor load imbalance and supports only 3.5 Gb
s. The CPU_Config implementation, which speeds up some of its partitions using the stream programming model, reduces the instructions per packet significantly but cannot overcome the overheads imposed by partitioning the application among six tiles: its maximum throughput is 4.85 Gb
s, more than LoadBal_Min but still slightly less than Data_Parallel.

Throughput Over Varying Memory Latency Values When each of our implementations is executed with varying remote memory latencies, however, throughput drops off for all implementations using multiple contexts per processor. Figure 4.2 shows the throughput of all five implementations for varying remote memory latencies. The x-axis shows memory latency values varying from 20 to 500 cycles, while

Figure 4.3: Throughput with Four Hardware Contexts

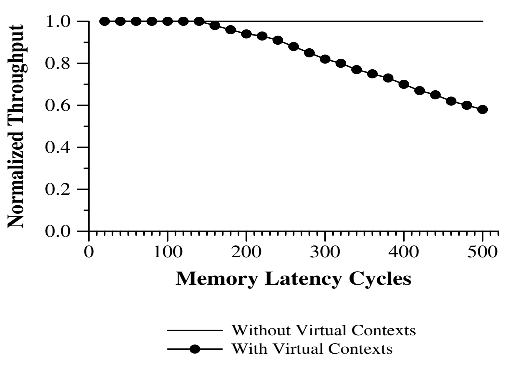
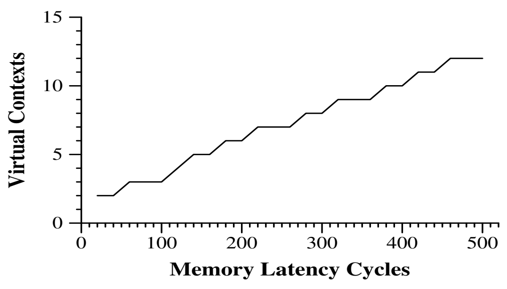


Figure 4.4: Virtual Contexts for Maximum Throughput



the y-axis shows the throughput values normalized to the Data_Parallel version executed at its maximum throughput. Each of the implementations using contexts to mask memory latency, Data_Parallel, LoadBal_Min, and LoadBal_Max, execute four independent contexts, maintained in hardware contexts, per processor. The two queue-based implementations, MemRef and CPU_Config, maintain their throughput across all memory latency values while the context-based implementations slow down as the contexts cannot mask all the memory latency and the processor begins to stall. The rate of change, however, varies among the context-based implementations. Data_Parallel throughput declines from 100% to 58% of the maximum and LoadBal_Max slows from 55% to 23%. LoadBal_Min throughput slows from 94% to 24%, the steepest drop of the three. Each of these implementations each execute multiple blocks of instructions which are terminated by a remote memory access that forces the context to pause; LoadBal_Min subdivides some of its blocks to balance the work across processors. As a result, some processors execute shorter blocks than those executed for Data_Parallel. Short blocks provide less work for the processor to perform between remote memory accesses, and as memory latency increases, LoadBal_Min processors quickly begin to stall. Although LoadBal_Max also subdivides its blocks, the blocks include the extra communication instructions and so provide the processor with more work.

Performance Effects of Virtual Contexts As Figure 4.2 showed, implementations which depend on multiple independent contexts to mask remote memory accesses can suffer slowdowns if there are not enough individual contexts to keep the processor busy. In our configurations above each context was permanently stored in a hardware context, making the hardware context a significant resource bottleneck. However, we can virtualize the hardware contexts by assigning more contexts to the processor than hardware contexts and allowing context state to be stored to and retrieved from local memory. Employing virtual contexts to support independent contexts is very similar to the use of queues in the task-parallel MemRef implementation. In each case the packet meta-data and a minimal amount of state is stored into a local memory buffer after executing a remote memory reference. When the reference completes, the packet meta-data and the associated state is eligible to be transferred back into a hardware context. There are two differences between the techniques: first, in MemRef the packet and state are transferred from one processor to the local memory buffer of another processor, where virtual contexts move between the processor and its own local memory. Second, a processor in MemRef executes the same code block for each packet it pops from its input queue. A processor executing the Data_Parallel application will include the address of the next instruction as part of the context state and use it to begin executing when the packet is restored to a context. Figure 4.4 shows the number of virtual contexts required to maintain maximum throughput for the Data_Parallel implementation. Remote memory latency values from 20 to 500 cycles are laid out on the x-axis and the total number of virtual contexts are shown on the y-axis. The virtual contexts required by Data_Parallel increases in a linear, if step-wise fashion, from two to twelve as memory latency grows to 500 cycles.

While virtual contexts can improve packet throughput, they also have secondary effects. Specifically, the additional concurrent contexts can conflict for processor resources and increase packet latency and jitter, while copying contexts into and out of local memory increases the local memory port accesses. We will report on performance metrics for the Data_Parallel implementations using a varying number of virtual contexts with 160-cycle remote memory latency: the remote latency value is a reasonable value for our system at this time, but it is not intended to be entirely representational: rather, we hold the parameter fixed to explore the effects of varying the number of virtual contexts.

We can model several performance metrics using characteristics of the executable. The work performed by a processor to handle a single packet can be viewed as a series of instruction blocks, each terminated by a remote memory reference. When a context stalls after making a remote memory reference the processor will execute instructions from the instruction block of another context, or stall if no other contexts are ready to execute. Taking the execution of a single packet as a frame of reference, we can predict the number of processor stall cycles as well as packet latency, packet jitter, and the local memory access frequency based on the number of virtual contexts. We can also predict the number of processor stall cycles introduced using a fixed number of hardware contexts. We observe that in our system each packet will be processed using the same instruction blocks performed in the same order, each block has very little execution variation, and the virtual contexts are scheduled in strict round-robin fashion. As a result the individual instruction blocks are executed in order; first each context executes block 0, then each context executes block 1, etc. Our models predict the simulated performance of our system to within 5%.

$$S = \sum_{i=0}^{n} \begin{cases} M - (b_i * (V-1)) & : \quad M > ((b_i * (V-1)) \\ 0 & : \quad M <= ((b_i * (V-1)) \end{cases} \tag{4.1}$$

The amount of processor stall is governed by three factors: the total number of virtual contexts and the size of the individual instruction blocks, which together represent the work available to the processor, and the remote memory latency: the model given in equation 4.1 articulates the relationship between the parameters. S represents the number of cycles the processor stalls during the processing of a single packet, M is the remote memory latency, V is the number of virtual contexts on the processor and b_i is the number of instructions in the ith instruction block. When a virtual context incurs a remote memory access after executing instruction block i the processor will stall when the remote memory latency M is longer than the work available to the processor, that is the execution of instruction block i by all the remaining virtual contexts. The number of virtual contexts required to eliminate processor stall is determined by the shortest instruction block.

Two applications with the same number of total instructions and remote memory references executed on the same number of virtual contexts can have very different performance. Table 4.2 gives an example of two such applications: we have modelled the execution of each application on a single processor with remote memory latency set to 160 cycles, using a varying number virtual contexts. The processor stall model results are compared against simulated execution in Figure 4.6 and Figure 4.7. *Equal* requires only four virtual contexts to execute with less than 5% processor stall where *Unequal*, whose shortest instruction blocks are much smaller than the Equal block sizes, requires nine for the same performance. Although the smallest block determines the maximum virtual contexts required, all the blocks contribute to the shape of the performance curve.

Our DiffServ application is profiled in Table 4.2. DiffServ has three instruction blocks of varying sizes, executing a total of 240 instructions. Figure 4.8 shows the processor stall for DiffServ executed with the remote memory latency set at 160 cycles, using a varying number of virtual contexts. Although DiffServ will require five virtual contexts to eliminate processor stall, fewer virtual contexts may not increase the processor stall significantly. At four virtual contexts the processor stall is 2.1% and at three virtual contexts it is 8.1%. With only two virtual contexts there is a sharper stall increase to 30.5%.

We have performed the analysis for this study using a system with very little variation, which allows us to clearly identify the performance factors and express them as equations. We further isolated the effects of instruction block size using abstract applications and demonstrated that while short instruction blocks require more concurrent virtual contexts, large size differences between instruction blocks increase resource conflicts and packet latency.

Although lack of execution variation helps us to identify and understand the performance of our system characteristics, we need to look ahead to see how our conclusions here will apply to a more dynamically variable system. Using data parallel application implementations simplifies the support for execution variation by leveraging the strength of programmable hardware. Multiple execution paths can be provided to packets just by using branches in the code, and each processor can continue to perform useful work on independent packets and maintain its efficiency. Applications which exhibit runtime variations could vary the number of instructions executed between remote memory references, or execute a varying number of instruction blocks per packet. For the purposes of performance modeling we can define the application

Application	Block 0	Block 1	Block 2	Block 3	Total Instructions	Variation
Equal	50	50	50	50	200	0
Unequal	10	10	50	130	200	0

as a series of block *slots*, each slot acting as a placeholder for the range of instruction block sizes which can execute in that position. An application will have n slots where n is the maximum number of blocks which can be executed for any packet, and each slot has a frequency associated with it denoting what percentage of packets use it. The blocks associated with a block slot may represent different execution paths through the same task code, or distinct tasks. Table 4.2 profiles three applications. The first, *Static*, is a completely static application with four block slots, each with a single possible instruction block size. Every packet uses all four slots, which means that every packet will have the the blocks A, B, C and D executed. The *VariableBlock* application has the same number of block slost for every packet but the instruction block size in each slot can vary: a packet can have blocks A, D, G, and J executed, or one of 63 other block combinations. The third application *VariableSlot* has four block slots which are executed for every packet, one slot which is executed for 15% of the packets, and one slot which is executed for 5% of all packets. A packet may have instruction blocks A,B,C, and D executed, or A,B,C,D, and E, A,B,C,D, and F, or A,B,C,D,E, and F.

We can model the execution of the variable applications by changing the term representing the work available to the processor. In the case of VariableBlock the b_i term will represent the average of the possible block sizes assigned to slot i, weighted by their execution frequency. The average block size for each slot in VariableBlock is the same as the block size for the same slot in the Static application, allowing us to directly compare their performance. For VariableSlot the packets will have a varying number of instruction block slots executed, which introduced two changes to our model. First, the concurrent packets may not be executing the same instruction slot: the instruction block sizes will reflect the average of all possible instruction blocks, weighted by the execution frequency of the block within the slot and the execution frequency of the slot itself. Second, as the processor stall calculation is based on the processing of a single *reference packet*, the stall contributions from each of the slots executed for the reference packet are weighted by the slot execution frequency. Our revised equation 4.2 introduces F_i, the execution frequently of slot i, and it uses the value b_a, the weighted average block size, in place of the size of a specific block.

$$S = \sum_{i=0}^{n} \begin{cases} (M - (b_a * (V-1))) * F_i & : \quad M > ((b_a * (V-1)) \\ 0 & : \quad M <= ((b_a * (V-1)) \end{cases} \tag{4.2}$$

The Static and VariableBlock applications will have very similar processor stall using a fixed number of virtual contexts, while VariableSlot will have much less stall. Figure 4.9 shows the modelled processor stall for each of the three applications, executed with remote memory latency set to 160 cycles, across a varying number of virtual contexts. The VariableBlock application has exactly the same stall as Static: in both cases the processor will have, on average, the same amount of work available to it. The VariableSlot has much less processor stall for a fixed number of virtual contexts and requires only six contexts to eliminate stall compared to seventeen required by Static and VariableBlock. In VariableSlot long instruction blocks can be scheduled concurrently with short ones, smoothing out the variations in the available work.

The accuracy of modelling processor stall using weighted average block sizes will be limited by asymetric nature of the performance metric. The VariableBlock application whose four concurrent contexts are each executing a block from slot 0 will execute an average of ten instructions each: for example, two contexts may execute ten instructions, one may execute five instructions and the last may execute fifteen instructions. There will be some instances where all four contexts execute only five instructions each, and statistically those instances will be offset by cases where each context will execute fifteen instructions each, so that the average block size remains ten instructions. However, the processor will accrue stall cycles in the first case which are not offset by the second case, a variation which our model will not account for. As a result, the maximum number of virtual contexts required, as reported by the model, are not guaranteed completely eliminate processor stall. To make that guarantee the variable implementation must be modelled with the assumption that the contexts will execute the shortest block sizes concurrently, as in the static model.

Using virtual contexts decouples the number of concurrent contexts from the number of hardware contexts required to support the maximum packet throughput. We can create a formula to predict the stall based on the number of hardware contexts

Figure 4.5: Stall Percentages: Virtual Contexts

Figure 4.6: Equal

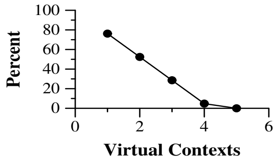
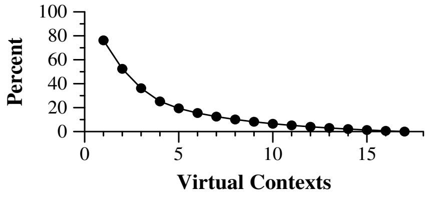


Figure 4.7: Unequal



Application	Block 0	Block 1	Block 2	Total Instructions	Variation
DiffServ	46	82	132	240	6

Figure 4.8: Stall: Virtual Contexts

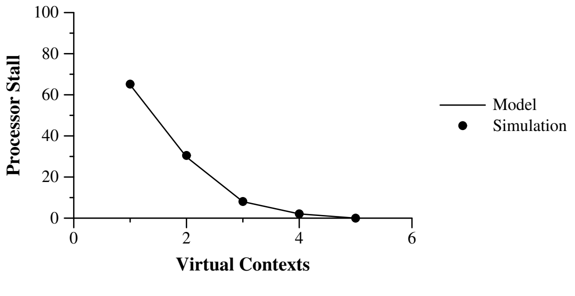
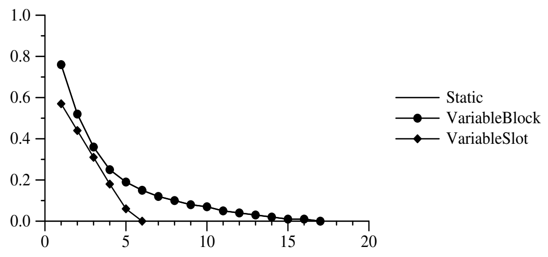


Figure 4.9: Modelled Dynamic Stall: Virtual Contexts



used, assuming the execution is performed with enough virtual contexts to support zero stall cycles. Each hardware context will cycle between two states:

1. live: holding a virtual context which is ready to execute an instruction, and
2. copying: holding a virtual context which is being copied into or out of local memory.

With one hardware context, the processor will stall whenever the hardware context begins copying because there is no other hardware context to execute instructions. If the processor uses i hardware contexts, it can begin executing from HC_1 as soon as HC_0 begins copying, etc. until $HCi - 1$ begins copying. If at that point HC_0 is not done copying, the processor stalls.

$$S = \sum_{i=0}^{n} \begin{cases} T - (b_i * (H-1)) & : \quad T > ((b_i * (H-1)) \\ 0 & : \quad T <= ((b_i * (H-1)) \end{cases} \tag{4.3}$$

In equation 4.3 S is the number of cycles the processor stalls, T is the time spent copying a virtual context to and from local memory, H is the number of hardware contexts used and b_i is the number of instructions in the ith instruction block. A system with more dynamic variation would require changes similar to those discussed for equation 4.1.

Figure 4.10 shows the processor stall for a varying number of hardware contexts; DiffServ needs only two hardware contexts to keep the processor from stalling.

The packet latency will include the cycles required to execute each of the instructions blocks, the memory latency cycles, and the cycles spent waiting for access to the processor. Packet latency is therefore also a function of the number of virtual contexts executing on the processor, which can conflict with each other for processor access.

$$L = \sum_{i=0}^{n} b_i + n * M + \sum_{i=0}^{n} \begin{cases} (b_i * (V-1) - M) & : \quad M < (b_i * (V-1)) \\ 0 & : \quad M >= (b_i * (V-1)) \end{cases} \tag{4.4}$$

In equation 4.4 L is the number of cycles of packet latency, M is the remote memory latency, V is the number of virtual contexts on the processor and b_i is the number of instructions in the ith instruction block. The packet latency is divided into two parts; the first part includes the instructions executed and memory latency cycles incurred on behalf of the packet, and the second part calculates the number of cycles the packet is blocked by other packets. A packet will be blocked if the remote memory latency is less than the other work available to the processor, which is the execution of instruction block i by all the remaining virtual contexts.

The percentage of processor stall cycles is calculated as S / L.

Large block size deltas will generate contention as the small block sizes increase the number of virtual contexts required and the large block sizes add to contention. Figures 4.12 and 4.13 compare the model predictions and simulation results for the packet latency when executing our Equal and Unequal applications with varying the number of virtual contexts. Unequal extends the packet latency by more than a factor of four before the processor stall is eliminated because the virtual contexts must wait for the executions of the longer instruction blocks to complete. As virtual contexts are added, packet latency will not begin to grow until at some point the processor has more work to do than it can finish within the remote memory latency: this will occur for the largest instruction block first. For Equal that point does not occur until it is executing five virtual contexts; Unequal begins to lengthen packet latency beginning with three virtual contexts. Additional contexts will generate contention from smaller and smaller blocks until only the smallest block is not generating contention. At that point packet latency grows linearly with additional context due to the round-robin context scheduling.

A system with more dynamic variation would require changes similar to those discussed for equation 4.1. For applications with a varying number of block slots the packet latency using a fixed number of virtual contexts should be diminished by the mix of small and large concurrent execution blocks: the smaller number of virtual contexts required to keep the processor busy should result in lower packet latency for corresponding processor stall.

Based on the results for Unequal above, we can predict that the long instruction block in DiffServ will cause the virtual contexts to conflict and extend the packet latency. Figure 4.14 shows that the packet latency is almost doubled when the application is executed on six virtual contexts.

Packet jitter, the variations in packet latency, is not related to the instruction block sizes, merely the sum of their variations. Jitter increases linearly with the number of virtual contexts. In the equation 4.5 J_T is the number of cycles of packet jitter, J_E is the variation in the number of instructions executed for each packet, and V is the number of virtual contexts on the processor. A system with more dynamic execution variation would have that variation represented in the J_E term; other terms could be added for other system variables such as the remote memory latency.

$$J_T = J_E * V \tag{4.5}$$

Figure 4.15 compares the model and simulation packet jitter measurements for our DiffServ application. When application instruction blocks have a variable number of executed instructions packets will have jitter even when only one virtual context is executing. Additional virtual contexts amplify packet jitter by adding the execution jitter from other packets to the time a packet spends ready to run.

Using virtual contexts increases the memory port traffic in two ways; first, it raises the frequency of read and write instructions executed, and second, virtual contexts are copied into and out of local memory. Equations 4.6 and 4.7 give formulas for calculating read and write memory port access frequencies. R and W are the number of register read and write instructions executed per packet, C is the number of context copies performed per packet, M_R is the number of reads performed to copy a context from local memory, M_W is the number of writes performed to copy a context to local memory, V is the number of virtual contexts and L is the total packet latency.

$$P_R = ((R + (C * M_R)) * V)/L \tag{4.6}$$

$$P_W = ((W + (C * M_W)) * V)/L \tag{4.7}$$

The number of memory port accesses increase with the additional virtual contexts until there are more virtual contexts than hardware contexts assigned to the processor:

Figure 4.10: Stall: Hardware Contexts

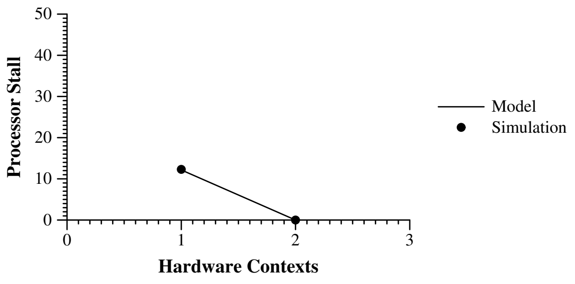


Figure 4.11: Packet Latency: Virtual Contexts

Figure 4.12: Equal

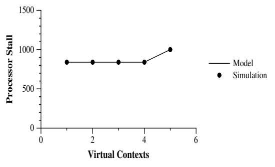


Figure 4.13: Unequal_2

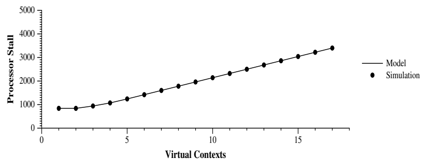


Figure 4.14: Packet Latency: Virtual Contexts

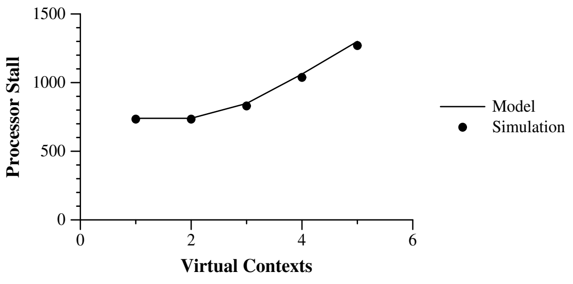
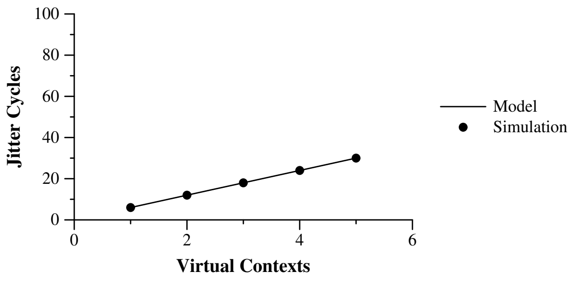


Figure 4.15: Packet Jitter: Virtual Contexts



then the access frequencies jump as the context copies add to the memory port traffic and continue to increase until the maximum throughput is reached. Figures 4.16 and 4.17 compare the model and simulation results for DiffServ execution, reporting the local read and write port access frequencies for varying numbers of virtual contexts. These results were generated using two hardware contexts, the number needed to prevent processor stall for this application. A more dynamic system could be modeled by adjusting R, W and C to reflect median values.

4.3 Performance and Packet Independence

In this work we assume packet independence for our system based on the common case high-bandwidth network traffic for a edge large router. Since task parallel implementations have occasionally been assumed to have better performance when processing dependent packets it is worth comparing the performance of data vs. task parallel implementations when packets are not independent. Because task parallel implementations naturally serialize the packet processing for each stage of the packet pipeline, it can continue to process dependent packets in parallel. The data parallel implementation could try to execute all n concurrent packets in the same stage on n processors; if the packets are related, it will be able to execute only one packet at a time. However, data parallel implementations do not have to have each processor core execute in lock step. The synchronization mechanism of the shared data structure will naturally stagger the packet processing across the processors: where the task parallel version would serialize the execution of the dependent stage on single processor, the data parallel version would serialize the execution of the dependent stage across all processors. The data parallel version and the task parallel version may both get the same performance in the face of dependent packet traffic. Figure 4.18 shows the throughput for our Data_Parallel and MemRef implementations processing network traffic composed of related packets: each packet requires read/write access to the same data object. The throughput for the two implementations is almost exactly the same. The advantage the task parallel version may have, which our results do not reflect, is data locality. Given data caching, executing a stage on a single processor would avoid copying the data structure between processors. Our implementations do not use data caching.

4.4 Conclusions

The results we report in this chapter demonstrate that a single decision, setting partitions to map an application to an array of processor cores, determines three outcomes: what advantages can be gained by the specific mapping, the processor load balance, and the communication overhead. Because the three rest on the same decision it is not possible to optimize for them all. The net result is that partitioning an application to gain significant benefit will probably incur significant overhead: conversely, attempting to manage overhead is likely to eliminate potential advantages. We conclude from this that in most cases applications should not be partitioned: each packet should be handled entirely on a single processor, using the processor array to exploit the data parallelism available in independent packets.

Using a data parallel implementation simplifies the packet handling execution but complicates the management of any resources external to the tiles. For example, processors executing a data parallel implementation can make simultaneous requests to an external memory port where a task parallel implementation could serialize all requests to that port, assuming data were partitioned among off-chip memory systems. Data parallel implementations will also require incoming packets to be distributed to all processors where the task parallel version may transfer them all to a single processor. If any temporal data locality is available, data parallel implementations cannot take full advantage of it where task parallel implementations may reduce some data copies and memory latencies. Similarly, a limited instruction buffer could introduce more instruction fetch latencies to a data parallel implementation than to a task parallel implementation.

Because our system offers software managed communication between processors, the communication required by our task parallel implementations placed additional pressure on the instruction fetch bandwidth. Although we could provide hardware support for inter-processor communication similar to the support we provide for moving data between the processor and local memory for context copies, the data transfer would be more complicated than the context switch, which is predicated on saving a small, fixed amount of state. Unless the inter-processor communication was similarly fixed, the hardware support must either manage a variable number of live registers or consistently transfer the data from all potentially live registers. In both cases the on-chip network traffic could increase significantly.

Although our experiments used statically generated task parallel implementations, we can apply our findings to dynamically partitioned and mapped applications. Dynamic partitioning can be used for applications without runtime variations, in which case the partitioning would be determined at runtime but remain stable. Apart from the initial partitioning overhead, the performance would be the same as a static partitioning of the same application: the dynamic partioning support merely serves as an automation tool. Dynamic partitioning can also be used to adapt the partition boundaries to variable execution: in this case the cumulative performance is the weighted sum of the performance of each individual implementation: the relationship between the setting the partitions and the associated benefits and costs remains the same. One difference between the static and dynamic mapping is in achieving a balanced processor load: while the statically mapped application must be dividing into equal partitions to achieve a balanced load, the dynamically mapped application can be partitioned at will and the partitions for each packet assigned to idle processors. If the partitions are restricted to certain processors (to manage the instruction buffers, for example) the performance will be similar to a statically partitioned application. If the partitions can be executed on any processor, most benefits of partitioning disappear and the execution begins to resemble a data parallel implementation with the additional runtime mapping overhead. The benefit which does not disappear with unrestricted partition mapping is reduced packet latency from exploiting intra-packet parallelism.

Application	Block Slot	Block	Size	Execution Frequency
Static	0	A	10	1.0
	1	B	10	1.0
	2	C	50	1.0
	3	D	130	1.0
VariableBlock	0	A	5	0.25
		B	10	0.50
		C	15	0.25
	1	D	5	0.25
		E	10	0.50
		F	15	0.25
	2	G	40	0.25
		H	50	0.50
		I	60	0.25
	3	J	100	0.25
		K	130	0.50
		L	160	0.25
VariableSlot	0	A	10	1.0
	1	B	10	1.0
	2	C	50	1.0
	3	D	130	1.0
	4	E	90	0.15
	5	F	150	0.05

Figure 4.16: Read Port Accesses: Virtual Contexts

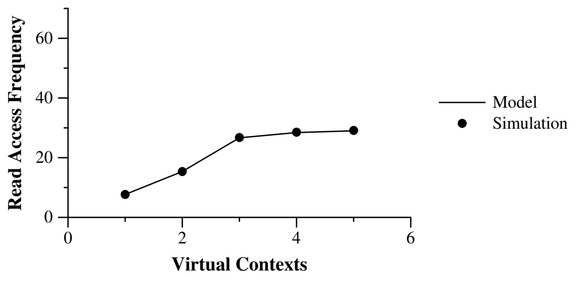


Figure 4.17: Write Port Accesses: Virtual Contexts

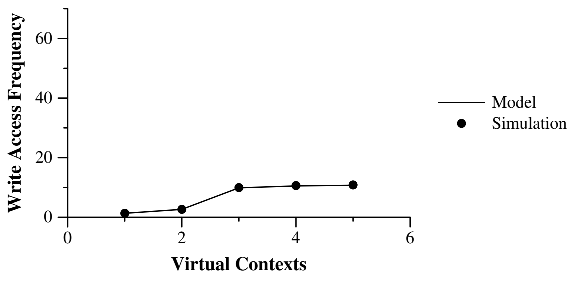
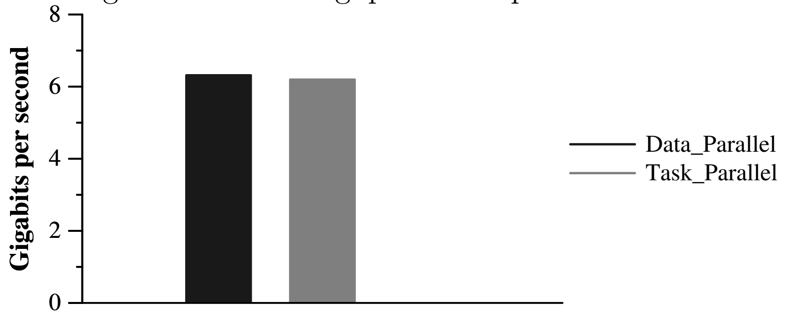


Figure 4.18: Throughput for Dependent Packets



Chapter 5

Reducing the Instruction Fetch Bottleneck

In the previous chapter we showed some performance advantages and limitations of mapping all of the packet processing work for a single packet onto the same processor to exploit data parallelism. When the processor stall has been minimized, the instruction fetch bandwidth is being used efficiently: further throughput improvements will require either using more efficient instructions or increasing the available instruction fetch bandwidth. In this chapter we explore creating some simple extensions to the MIPS instruction set architecture which can compress tasks into fewer instructions. We then compare two techniques for doubling the instruction fetch bandwidth using alternate configurations for our Smart Memories processor cores and examine how all these optimizations change the resource requirements and performance effects in the context of a data parallel implementation using virtual contexts. Lastly, we report on the branch behavior of our optimized executable and review a fine-grained scheduling algorithm for keeping the pipeline full.

5.1 Compressing the Executable

Our packet-processing application is composed of several short tasks which share the same basic structure:

1. Read in a data object.
2. Perform computation using most or all fields of the data object.
3. Update the packet meta-data and/or data object.

Our target edge router manages very large data structures stored in off-chip memory and the data object layouts are compressed to minimize object size in order to reduce the demands on the memory bandwidth and the data footprint. This basic packet processing execution profile gives us the opportunity to improve throughput by making the data struction handling more efficient: we do this by extending the instruction set architecture.

To read a data field using a conventional load/store interface, a task copies the field from the data object into a register using a load instruction. The MIPS IV instruction set architecture used to implement our workload allows a single load instruction to copy one, two, four or eight bytes of data into a register. If fewer than eight bytes are requested, the full eight bytes are read from local memory and the data masked and aligned correctly in the register. If the field is not aligned on byte boundaries it may take several instructions to move the data field into the register and align it properly. In addition to the instruction overhead of managing individual data fields, the demand on the bandwidth between the registers and local memory may be inflated since each load or store instruction will consume a fixed amount of bandwidth (e.g. 64 bits) regardless of the size of the data field (e.g. 5 bits). Optimizing this data handling can potentially improve the throughput and reduce the demand on local memory bandwidth for the entire application.

To streamline the execution of our application we extend the instruction set architecture to support moving contiguous data into multiple registers: an entire 64-byte data structure can be read in using a single instruction, and its data fields will be stored in the registers in their original layout. As a result, a single register may hold multiple unaligned data fields. To manage the compressed data efficiently we add instructions to perform simple computation on individual data fields contained within registers; we support logical operations and simple ALU operations. We do not support data fields which span registers. Each of the new instructions can read from or write to a restricted range of contiguous bits within one of its operand or result registers: the remaining registers are handled normally.

The new load instruction is encoded to specify the source address (which must be local to the processor), a single destination register, and the total number of target registers. The data will be loaded into a group of contiguous registers beginning with the specified target. Store instructions are encoded similarly. A memory operation using multiple registers is executed on one register at a time during multiple cycles, exactly as if multiple processor instructions had been issued.

A computation instruction using a bit range from an operand register is encoded to include a shift bit field and a field indicating the number of bits in the target range, each six bits long. The register value is shifted, a mask is constructed to extract the bit range from the register, and the operation is performed on the isolated bit range. If the instruction stored a value to a bit range in a register the computation would occur first, then the shift and mask. *@@@ diagram* A similar approach was used in extending the instruction set for the FLASH Multiprocessor, which supported bit field insertion and extraction instructions. *ref ISA spec*

Using the new instructions can introduce several sources of complexity to the architecture. First, adding fields to instructions can make the decoding process less regular. For this work we have verified that a 32-bit instruction can be encoded to specify our additional fields, but we do not specify the decode logic. Second, by using a single instruction to perform memory operations on multiple registers we may increase contention for the memory and register ports. In this work we schedule the memory ports of the processor, but we assume there are no restrictions on the register port use. Lastly, by offering instructions to extract and operate on bit ranges in a register we require the architecture to support a short chain of dependent operations. We accomplish this by adding an additional execute stage which is used to complete the ALU operation (when an operand uses a bit range) or the shift and mask (when the result uses a bit range).

The extensions proposed here are a combination of two existing techniques. The Intel IXP family of network processors *ref* support group register loads and stores: however, an individual data field must be extracted from one register into another in order to operate on it. Malik et al [27] proposed an enhanced ALU for general-purpose computing to streamline the execution of dependent instructions by collapsing them into single-cycle equivelants. In our work we focus on a restricted set of operations, targeted to our particular domain, which require only simple architectural support.

@@@Include pipeline figure, logic diagrams

5.2 Increasing Instruction Fetch Bandwidth

We can continue to improve packet processing throughput by increasing the instruction fetch bandwidth, an option supported by our reconfigurable Smart Memories processor cores. In chapter 4 one of our application implementations, CPU_Config, used microcoded instructions for portions of the application to schedule functional units in parallel. In this chapter we examine two methods for allowing a pair of MIPS instructions to be scheduled in parallel: in this way we can increase the instruction fetch bandwidth for the entire application using a single processor configuration and avoid the need to partition the executable. Our first dual-issue configuration supports VLIW instructions to exploit the instruction-level parallelism statically available in the executable. Each VLIW instruction is made up of two independent MIPS instructions: if no second independent instruction can be scheduled during a cycle the second half of the VLIW instruction is a 'nop'. Our configuration can issue a single in-order instruction from each of two contexts per cycle, a static variation on simultaneous multi-threading which originated in 1995 at the University of Washington [36]. Any conflicts for the memory ports and multiply or divide functional units are resolved at run time.

5.3 Methodology

The base implementation is modeled after the simple Data_Parallel implementation from Chapter 4. The classification algorithm has been changed slightly to accommodate the new memory interface; the iterative matching loop processing the element array within a node has been completely unrolled to manage the array stored in a group of registers. Each additional implementation uses an increasingly aggressive set of new instructions. The new instructions are hand encoded and written into the binary after it is generated, replacing a placeholder system call instruction.

1. The *Base* workload implementation uses MIPS ISA load/store instructions operating on single registers.

2. The *Reg* workload implementation uses load/store instructions operating on groups of up to eight contiguous registers.

3. The *Min* workload uses group register operations with instructions which extract bit fields and perform simple logical operations.

4. The *Med* workload uses group register operations with instructions which extract bit fields and perform simple logical and arithmetic operations.

5. The *Max* workload shows the performance improvement upper bound by using group register operations with instructions which extract bit fields and perform any operations.

6. The *Med_Data_Stall* workload uses the same instructions as Med, but the execution includes pipeline stalls caused by the additional execute stage.

Application implementations which use the extended instruction set increase the size of their context state which must be copied between the register file and local memory, since the registers hold complete data objects instead of their pointers. The context state written into local memory increases from seven registers for each copy to eleven registers which includes the packet meta-data object constructed and maintained by the application. When the context is copied back from local memory using a total of nineteen registers: the extra registers receive a copy of the remote data object requested before the context yielded the processor. If the memory request has not been completed when the context is copied back to the processor, only the original state is read in and the data object is copied in later.

Our experiments compare two dual-issue configurations against a single-issue configuration: all three execute our Med implementation using the extended ISA.

- *Base*: One instruction issued per cycle.
- *Static_ILP*: Two instructions issued per cycle from a single context.
- *Static_SMT*: Two instructions issued per cycle, one from each of two contexts.

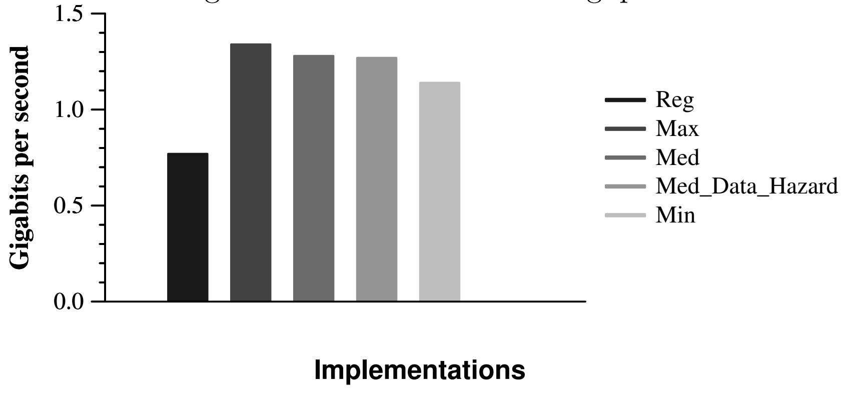
Static_ILP instructions are statically scheduled and conflicts for resources are resolved at compile time. Conflicts between Static_SMT instructions are resolved at runtime. The primary executing Static_SMT context has first option on the shared resources; however, because the workload uses the extended instruction set outlined in chapter 5, the memory ports are reserved for several contiguous cycles to support group register moves and either the primary or secondary context, or both, may stall waiting for memory port access. Neither context will be stalled by virtual context loads and stores (see Chapter 3).

5.4 Experimental Results

5.4.1 Maximum Throughput

All but one optimized implementation improves the maximum potential throughput of our application, executed using unrestricted local memory bandwidth. Figure 5.1 shows the maximum throughput of the five workload implementations using extensions to the instruction set; the y-axis measures the throughput normalized to the Base performance. The Reg implementation, which performs group register loads and stores but includes no new instructions to manipulate data fields, requires extra instructions to shift and mask the data fields before operating on them; this overhead slows the execution by 23% compared to the Base implementation. Although Reg uses more registers than the other versions, it does not need to spill register values to local memory; all of the performance slowdown is caused by the extra data manipulation instructions. The Max version, representing the performance improvement upper bound, has a speedup of 34%. Med, which combines data field extraction from a single register with simple logical and ALU operations, improves throughput by 28% while the Med_Data_Stall variation, which inserts a one-cycle pipeline stall

Figure 5.1: Normalized Throughput



reflecting data hazards caused by the extra execution stage, has equivelant speedup, indicated an insignificant number of pipeline stalls. Min, which does not require the extra execute stage, delivers a 14% speedup using logical operations combined with data field extraction.

Our packet-processing application is characterized by simple integer-based computation with frequent branches. Extending the ISA to provide more efficient instructions has reduced the total number of instructions but not the number of branches: as a result, there is limited statically-available ILP. Despite that limitation the Static_ILP implementation achieves a significant speedup. Although some processor resources are shared, Static_SMT leverages the simple computation to execute with near perfect speedup using the parallel ALUs. Figure 5.2 shows the maximum throughput of our Static_ILP and Static_SMT executions normalized to the maximum Single throughput. The Static_ILP speedup is 52% over Single while the Static_SMT speedup is 99%.

5.4.2 Virtual Context Performance Effects

All three strategies for reducing the instruction fetch bottleneck affect resource requirements and contention. In this section we analyze the performance implications of changing the number and size of executable's instruction block sizes.

Figure 5.2: Wide Issue Throughput

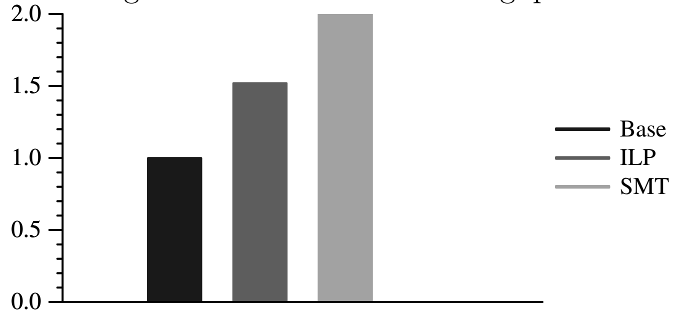


Table 5.1: **Application Instruction Blocks Sizes**

Application	Block 0	Block 1	Block 2	Block 3	Total
Base	31	59	103		193
Med (Single Issue)	27	24	46	53	150
Static_ILP	16	14	30	38	98
Static_SMT	27	24	46	53	150

Resource Requirements We can apply the same analytical models we used in chapter 4, expressed in the equations 4.1, 4.3, and 4.4, to understand how the extended ISA and VLIW instruction scheduling have changed the resource requirements. We can see from the instruction count breakdown in table 5.1 that the optimized Med implementation has several differences compared to the Base implementation. The Med implementation is optimized to execute in fewer instructions than Base, and each instruction block has been streamlined. Static_ILP further reduces the size of each instruction block. As our performance models indicate, when the total number of instruction cycles per packet is reduced it is the changes to individual instruction blocks which determine many performance effects. In addition to fewer instructions, Med also has one more instruction block than Base: because Med has a multi-word local memory reference which takes 8 cycles to complete, the context yields the processor. This yield breaks up a single long instruction block into two shorter ones. However, most models do not use the number of individual instruction blocks as a parameter. The increased number of instruction blocks will only change the memory port utilization by increasing the number of context copies per packet. All of the graphs in this section show results generated with memory latency fixed at 160 cycles, which gives an example of the relationship between available resources and performance for a reasonable latency value for our system. The results reflect execution with unlimited local memory bandwidth unless noted otherwise. Our models predict the simulated results with less than 5% error, with the exception of the hardware context model which has high prediction error for processor stall values below 5%.

Virtual and Hardware Contexts The Med and Static_ILP implementations have each reduced the number of cycles required to execute every individual instruction block. Because they optimize the smaller instruction blocks, we can predict they will increase the number of virtual contexts required to supply the processor with work. Since the difference is not large for Med (7 instructions, or less than 25% of the original size) the number of additional virtual contexts required for is likely to be relatively small: Static_ILP has a larger reduction (10 more instruction cycles, or 42% of Med) and its virtual context increase is likely to be correspondingly greater.

Because Static_SMT does not change the instructions cycles required per packet, the individual context behavior should not change. Static_SMT behavior should be similar to that of two single-issue processor cores combined: the total number of virtual contexts should double that of Med. Figure 5.3 shows the processor stall for the Base, Med, Static_ILP and Static_SMT implementations varying the virtual contexts per processor. All results were collected with the remote memory latency set at 160 cycles. Static_SMT processor stall is collected individually for the two issue slots: we report the average value. The curves generated by the analytical models are overlaid with the results generated by our simulator. Med requires eight virtual contexts to eliminate processor stall, Static_ILP needs thirteen and Static_SMT needs sixteen compared to seven virtual contexts for Base, consistent with our expectations. Med and Static_ILP increase the context copy frequency for their contexts by reducing the instruction block sizes: they also increase the context copy cycles because the context state has grown. They may require more hardware contexts than Base to pipeline context copies. Static_SMT should require the same number as two single-issue processors executing Med. Figure 5.4 shows the processor stall resulting from varying the number of hardware contexts with the simulation results for each implementation overlaid on the curves generated by the model. The Med and Static_ILP versions each require three hardware contrexts to eliminate processor stall, one more than the Base version, but with two hardware contexts Static_ILP has more processor stall (11.5%) than Med (1.6%). Static_SMT requires six hardware contexts, twice that of Med. However, all three optimized versions can execute with less than 5% processor stall using three hardware contexts.

Packet Latency and Packet Jitter Based on our models, the larger the difference between instruction block sizes the more contention between contexts is generated for a fixed number of virtual contexts. We can expect that Med and Static_ILP, which each use fewer cycles per instruction block than Base, will result in lower packet latency for a fixed number of virtual contexts due to fewer total instruction cycles per packet: they will also generate less contention between contexts since the difference between the instruction block sizes is less than for Base. It is not clear, however, whether they will have lower packet latency for a fixed processor stall value: Med and Static_ILP both need additional virtual contexts to achieve the same processor stall as Base, and the extra contexts could introduce enough contention to increase the packet latency. Static_SMT should execute with the same packet latency as a single-issue processor executing Med using half the virtual contexts. Figure 5.5 shows the packet latency using varying numbers of virtual contexts for each implementation. Each packet latency value for both implementations, generated using our simulator, is compared to the curves generated by our latency model. The results are consistent with our observations: at six virtual contexts, Base executes with 1163 cycles of packet latency while Med executes with only 965, Static_ILP with 768. Static_SMT executes with 736 packet latency cycles, equal to Med using three virtual contexts. When each is executed with zero processor stall, Base has 1351 packet latency cycles and Med has 1200 and Static_ILP has 1274: the additional virtual contexts do not contribute enough contention to overcome the shorter processing time compared to Base. Static_ILP does generate enough contention to increase its packet latency compared to Med despite its shorter instruction blocks. Static_SMT executes with 1200 packet latency cycles, equal to Med.

Packet jitter, which we model using the formula in 4.5, does not depend on the instruction block size. However, the optimizations have dropped the execution variation for Med and Static_ILP by half, and this reduces the effect of concurrent virtual contexts on total packet jitter. We can expect that Med and Static_ILP jitter will be half that of Base for any fixed number of virtual contexts. Static_SMT jitter will be slightly elevated compared to Med executed on a single-issue processor due to the runtime resource conflict resolution. Figure 5.6 compares the packet jitter results for the our four implementations generated by the model and simulator and confirms our estimates.

Local Memory Traffic The changes we have made to reduce the instruction fetch bandwidth performance bottleneck also make multiple changes to the local memory bandwidth demands. Our ISA extensions allow us to copy compressed data objects into and out of an array of registers, reducing not just the number of load and store instructions executed per packet, but also the data transferred by the instructions across the local memory ports. Figure 5.7 shows the read and write accesses executed by Med, normalized to the total read and write accesses executed by the Base implementation. Med executes 28% of the register reads by load instructions, and 75% of the register writes by store instructions. This reduction in memory port traffic is offset by two more differences. First, more context state is now copied between the processor and local memory and second, the number of context copies per packet has increased from three to four. Although eight of the additional context state registers are assigned to hold a copy of a remote data objects, each data object is copied into and out of the context registers only once: those data transfers replace the loads and stores executed in the Base implementation. The object is transferred into the registers as part of the context copy into a hardware context, and if it has been modified it must be explicitly written out to memory: the persistent data object is not included in the context state when a context is copied from the processor to local memory. The packet meta-data object is included in the context state in every context copy: its data is moved in and out of registers multiple times and increases the total number of memory accesses per packet. Due to the two changes to the context copy register operations per packet, the Med implementation executes 520% the number of context copy register reads from local memory and 214% the number of register writes compared to Base.

In addition to the increased number of register operations per packet, the ISA extensions reduce the total number of instructions executed per packet and further increases the frequency of memory instructions and context copies. Figure 5.8 shows the frequency of memory instruction register operations as the number of virtual contexts increases: at 34%, the total Base frequency exceeds that of every other implementation. The context copy register operations for Base remain low at less than 20% while the optimized implementations range from 80% for Med to almost 160% for Static_SMT. Figure 5.10 shows the frequency of all read and write accesses: at 95%, the Med frequency at maximum throughput is almost double that of Base at 53%. Static_ILP and Static_SMT have 145% and 191% access frequencies, respectively: Static_ILP increases the frequency of accesses per packet by further reducing the number of instruction cycles per packet while Static_SMT increases the total frequency of accesses without changing the frequency per packet.

The access frequencies reported above do not reflect any bursty behavior which might cause contention and increase the number of memory ports required to prevent processor stall. In Figure 5.11 we report the processor stall for each implementation varying the number of read/write memory ports. Each implementation is executing with the maximum number of virtual and hardware contexts required to support its maximum throughput. The Base implementation needs only one memory port to prevent any processor stall while the Med implementation suffers 4% stall using one memory port. Static_ILP and Static_SMT both execute with approximately 30% processor stall using one memory port and zero stall with four memory ports. In both cases a second memory port drops the processor stall to below 5%. Since the throughput for Static_SMT is significantly higher than that of Static_ILP and their memory port use is quite similar, Static_SMT uses its memory ports much more efficiently than Static_ILP.

[MARK: what units? percentages?] [MARK: do reads and writes add up to 100 percent port utilization (60 + 40)?] [MARK: reads conflict with writes right?]

5.4.3 Branches and Pipeline Stalls

The results reported so far are based on execution with no pipeline stalls caused by branches. Our application domain is characterized by frequent branches and our optimized application implementation Med reduces the total number of instructions executed per packet while keeping the number of branches the same as Base. Figure 5.12 measures the frequency of branch instructions during the execution of the workload on a single packet: Med executes with 27% branch frequency, where Base branch frequency is 21%. Because Med branch frequency is higher, it has more potential branch-related pipeline stalls to penalize the performance for the optimized implementations. Figure 5.13 shows the throughput of the Base and Med implementations executed with two-cycle stalls inserted after each conditional branch. Each implementation is normalized to the execution of the same implementation executed

Figure 5.3: Processor Stall: Virtual Contexts

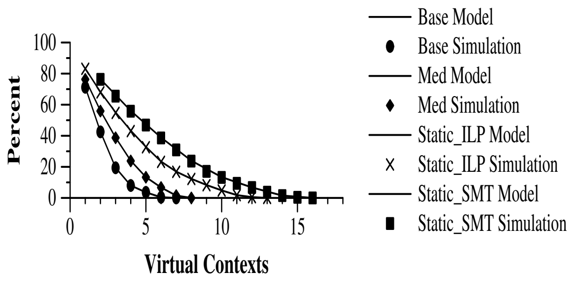


Figure 5.4: Processor Stall: Hardware Contexts

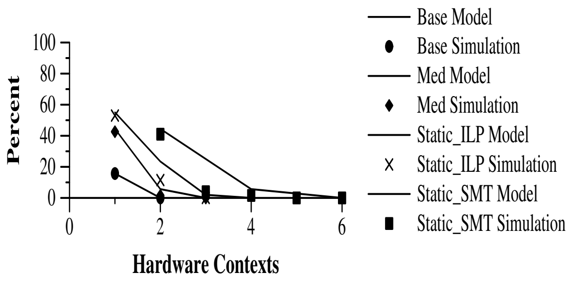


Figure 5.5: Packet Latency

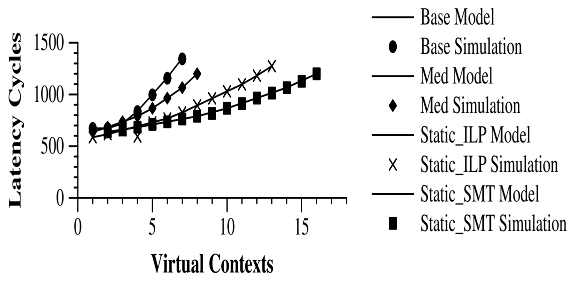


Figure 5.6: Packet Jitter

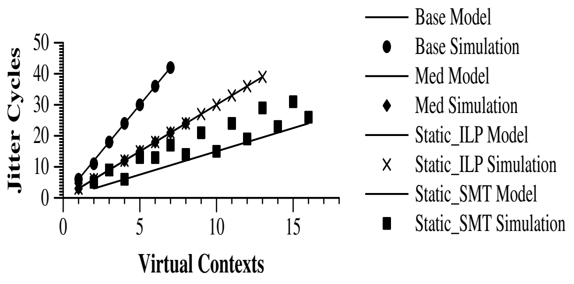


Figure 5.7: Register Read and Write Operations

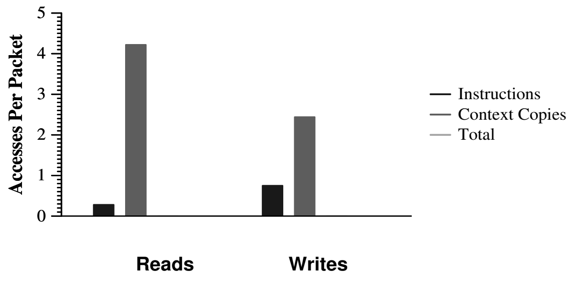


Figure 5.8: Load/Store Access Frequency

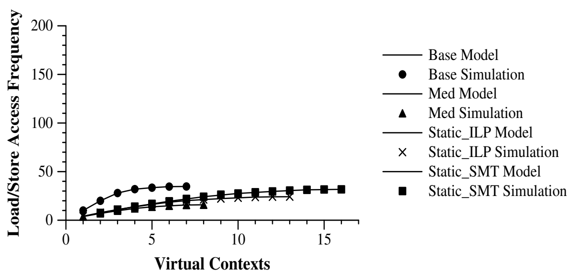
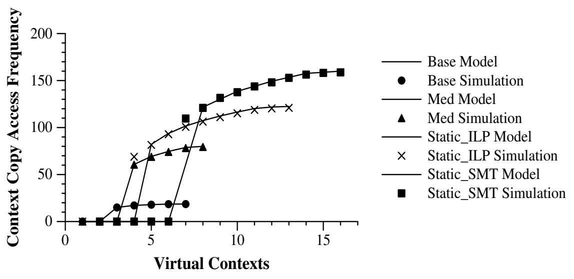


Figure 5.9: Copy Access Frequency



with no branch-related stalls (e.g. Base With Stalls is normalized to Base). The slowdown of Med is approximately 30%, compared with the 10% slowdown for the Base implementation: managing pipeline stalls can be a significant factor in maintaining the potential throughput for our optimized application.

$$V = |1.0 - |0.5 - t| * 2| \tag{5.1}$$

The branch behavior for our application is relatively unpredictable, as we show in figures 5.14, 5.15, and 5.16. Figure 5.14 reports the number of conditional branches executed per packet as a percentage of the total number of branches executed per packet; it shows that almost all branches in our workload are conditional. Figure 5.15 measures the number of conditional branches taken per packet as a percentage of the total number of conditional branches per packet; it shows that the conditional branches are taken about half the time. Figure 5.16 shows the variability of each individual branch in the executable, computed using equation 5.1: t is the frequency the branch is taken and V is the variability of the branch. The individual branches are laid out on the x-axis and the variability is measured on the y-axis. A branch which is taken half the time will have a variability of 1.0: a branch which is always taken will have a variability of 0.0, as will a branch which is never taken. Variability of 0.5 denotes a branch which is taken 25% and not taken 75% of the time, or vice versa. While many branches are very stable, approximately 25% of conditional branch results vary at least 25% of the time. Taken together, these results show that the branch behavior of our workload is significantly variable.

We can keep the instruction pipeline full by scheduling a context switch every cycle, already implemented in the Honeywell 800 (1958), the CDC 6600 PPU (1964), the Denelcor HEP (1978) and in the Tera MTA (1990), among others. We will compare execution of our optimized application implementation Med using this fine-grained scheduling algorithm against the same application executed using our original coarse-grained algorithm with no pipeline stalls inserted: this case represents the perfect behavior of other branch management techniques, and allows us to directly compare the requirements and performance effects of virtual and hardware contexts.

Figure 5.10: Combined Access Frequency

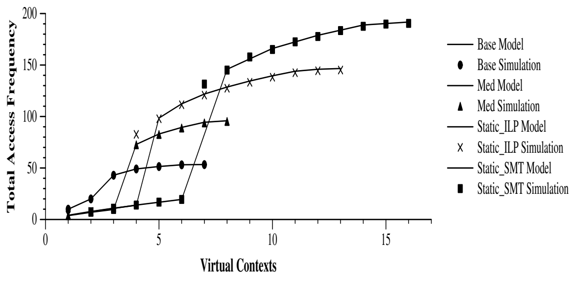


Figure 5.11: Processor Stall: Memory Ports

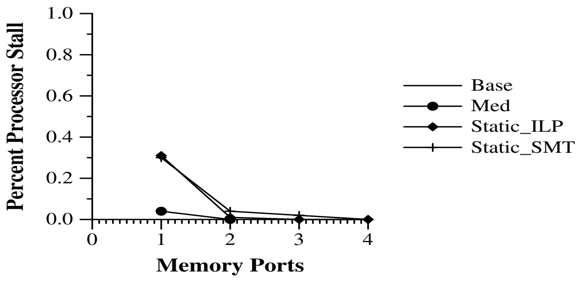


Figure 5.12: Branch Frequency

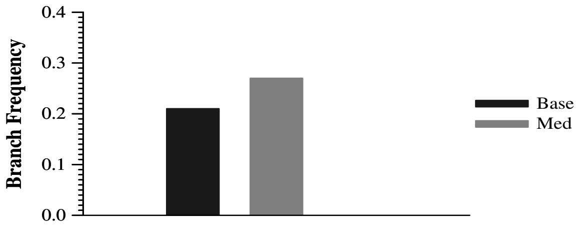


Figure 5.13: BranchStall_Throughput

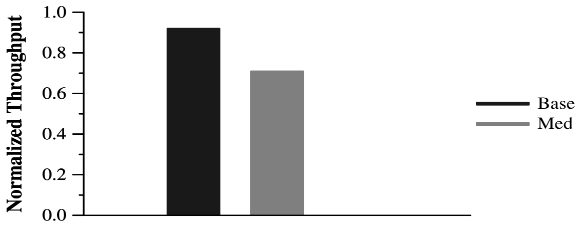


Figure 5.14: Conditional Branches

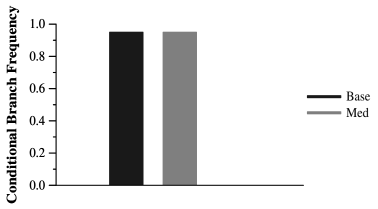


Figure 5.15: Conditional Branches Taken

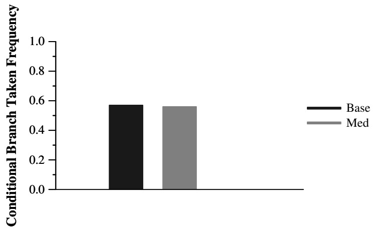
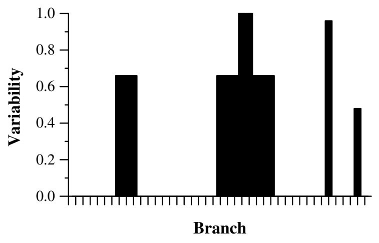


Figure 5.16: Conditional Branch Variability



1. *Coarse*, which performs a context switch after the executing context makes a remote memory reference or yields the processor, selecting contexts in strict round-robin order, and

2. *Barrel*, which performs a context switch after a single instruction, selecting contexts in strict round-robin order from the *execute pool*, a set of the contexts stored in hardware. If a context is not ready to execute it is removed from the execute pool and replaced by another context. If no other context is ready, the processor is idle during its execution slot in the rotation. The size of the pool is dictated by the length of the longest possible pipeline stall: we execute Barrel with a pool size of three to hide the two-cycle branch delay.

The intuition behind the formulas we have used based on our original scheduling algorithm holds for execution using the Barrel algorithm, but existing models do not reflect the performance effects of fine-grained switches among a pool of executing contexts. In this chapter we use augmented formulas which model the the regular scheduling we use in Barrel. The new variable P represents the size of the context pool used for fine-grained scheduling:

$$S = \sum_{i=0}^{n} \begin{cases} b_i * P - V & : \quad P > V \\ 0 & : \quad P <= V \end{cases} + \sum_{i=0}^{n} \begin{cases} M - (b_i * (V - P)) & : \quad M > ((b_i * (V - P)) \\ 0 & : \quad M <= ((b_i * (V - P)) \end{cases} \tag{5.2}$$

$$S = \sum_{g=0}^{P} \left\{ \sum_{i=0}^{n} \begin{cases} (T - (b_i * P * B_g))/P & : \quad T > ((b_i * P * B_g) \\ 0 & : \quad T <= ((b_i * P * B_g) \end{cases} \right. \tag{5.3}$$

$$L = \sum_{i=0}^{n} b_i * P + n * M + \sum_{i=0}^{n} \begin{cases} (b_i * (V - P) - M) & : \quad M < (b_i * (V - P)) \\ 0 & : \quad M >= (b_i * (V - P)) \end{cases} \tag{5.4}$$

$$J_T = (J_E * P) + (J_E * (V - P)) \tag{5.5}$$

Each of these formulas can replace their earlier counterparts from chapter 4 in our previous experiments without introducing additional error by setting P to 1.

Fine-grained scheduling increases the number of hardware and virtual contexts required to prevent processor stall because the execute pool allows contexts to make progress concurrently and cluster their remote memory references. Figure 5.17 shows the processor stall for Coarse and Barrel varying the number of virtual contexts supported by the processor. The Coarse version requires only eight contexts to completely eliminate processor stall while Barrel needs twelve.

Our fine-grained scheduling algorithm also increases the number of hardware contexts required to prevent processor stall. It must execute with a minimum of three hardware contexts to store the contexts in the execute pool: additional hardware contexts buffer the contexts being copied between the processor and local memory. Figure 5.18 shows the processor idle for our two scheduling algorithms varying the number of hardware contexts: although our model remains consistent with the simulation results at the end points, the middle values do not reflect the simulated results. The model is calculating the performance based on the worst case, when the largest number of contexts cluster their copies. In practice context copies can become staggered over time: our simulation results show the processor stall approaches zero starting with one additional hardware context for each algorithm.

Packet latency is increased by the rotation among the contexts in the execute pool, introducing stall cycles for an individual context during its execution of an instruction block. However, the execute pool leaves fewer contexts to conflict for access to the processor. Figure 5.19 shows the packet latency for varying numbers of virtual contexts executing our two algorithms. Although Barrel starts with significantly higher packet latency than Coarse, the results draw closer as the context count approaches eight (the maximum required by Coarse to eliminate processor stall): the Coarse version is adding packet latency faster than the fine-grained versions. However, as processor stall is eliminated Barrel finishes with 1825 packet latency cycles, 153% longer than Coarse.

Packet jitter is amplified by fine-grained scheduling, since a context spends $P-1$ cycles idle for every instruction it executes, as well as increased by variations in scheduling. Figure 5.20 shows the packet jitter for the three scheduling algorithms executed with varying virtual contexts. Barrel, which performs a context switch on every cycle, introduces the most jitter when the number of contexts is not evenly divisible by the scheduling pool size. When Barrel is executed with three, six, nine or twelve contexts for an execute pool of three, the jitter remains low and our model predicts it with up to 16.7% error. For other values the error rises to over 300%.

Barrel does not change the throughput or the total number of context copies executed per packet: as a result, the read and write access frequencies remain unchanged from the Coarse scheduling algorithm.

5.5 Conclusions

We showed in chapter 4 that a data parallel implementation supported by virtual contexts can use the processor instruction fetch bandwidth efficiently. A natural avenue for additional performance improvement is to optimize the instructions executed to process packets. In this chapter we identified characteristics of packet-processing tasks which could be leveraged to compress the application executable into fewer instructions: short application tasks executing integer-based computation using data fields packed into a small footprint can take advantage of our extended instruction set architecture to manage the data objects using fewer instructions. We also employed a processor configuration supporting VLIW instructions and further reduced the number of cycles required to execute the instructions for each packet.

Reducing the instruction cycles per packet increases the demands on local memory bandwidth: copying the context state between the processor and local memory represents a significant part of the increased demand. The context state footprint, and in particular the packet meta-data object created and used by the application for the life of the packet, should be carefully optimized. Extra processor yields, if they have been inserted to manage packet latency, should be balanced against the memory port utilization.

Using dual issue to exploit ILP increased our application's packet throughput by changes the instruction blocks for individual packets. The result was decreased contention between contexts for the processor and increased contention for the hardware contexts and memory ports. Using dual issue to execute instructions from independent contexts does not change the instruction block sizes: since the throughput improvement comes from executing independent contexts in parallel, demands on shared resources will become staggered, smoothing out bursty accesses and requiring fewer resources to maintain processor stall below a small percentage.

We do not model register ports but we can make some inferences based on our experiments. Register port conflicts could arise from either the context copies or the multi-register data structure operations. Given a set of read and write register ports, context and data structure operations could easily conflict with the register operations of the executing context. Those operations could yield to the executing context: starvation would not occur since eventually the processor would not have any contexts to execute, but it could introduce processor stall by not allowing processor stall could result. The register file architecture can also be designed to support both pipeline and background port accesses by adding register ports. Because the multi-register memory operations use registers which are part of the context state, the complexity introduced by multi-ported register files could be reduced by partitioning the registers into two files. The registers used for context state could be allocated to a file with one or more additional register ports while the remaining registers would belong to a register file with enough ports to support the pipeline accesses.

Figure 5.17: Processor Stall: Virtual Contexts

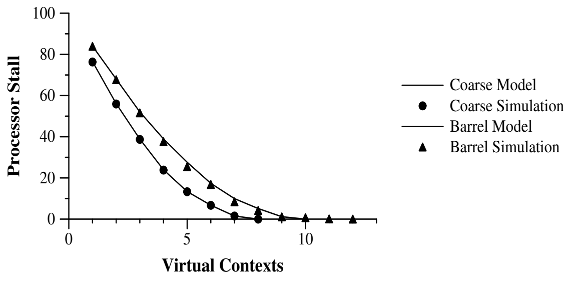


Figure 5.18: Processor Stall: Hardware Contexts

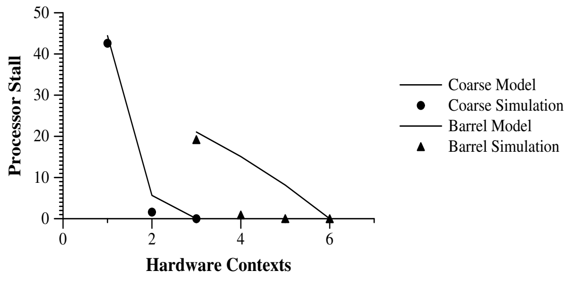


Figure 5.19: Packet Latency

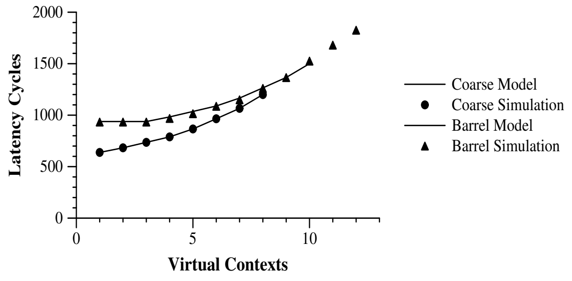
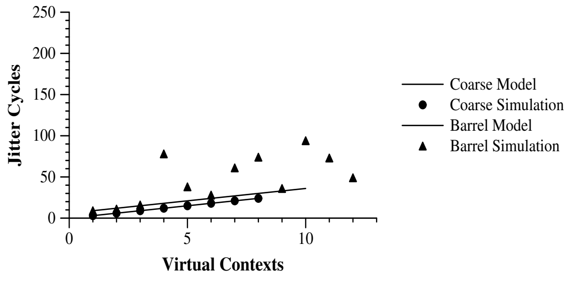


Figure 5.20: Packet Jitter



Chapter 6

Conclusions

Our motivation for the work we present in this dissertation is the growing need for heterogeneous packet processing services performed for high volume network traffic: filling this need requires implementations which are both fast and flexible. Designing packet processing systems to exploit parallelism is a natural avenue for this application domain. Packet-processing for large edge routers requires simple integer-based computation which is frequently memory-bound, and packets handled concurrently are predominantly independent of each other. We have performed a series of experiments executing multiple implementations of a widely-used packet processing application on a reconfigurable chip multiprocessor using network traffic made up of independent packets: using this system we have explored techniques for mapping work to an array of processor cores and for managing the resources of an individual processor. We have shown that characteristics of our application domain can be leveraged to increase the efficiency of programmable hardware and manage the contention for hardware resources.

We have identified key application and system characteristics which act as performance parameters and used them to evaluate the techniques we explored in this dissertation and extrapolate beyond our experimental analysis. Although the maximum packet throughput available to a particular application is determined by the total number of instructions executed per packet, packet-processing applications in our domain are typically characterized by a small amount of computation punctuated by long memory latencies. As a result, the number of instructions between remote memory references becomes an important performance parameter determining resource efficiency and contention for each of the techniques we have explored.

6.1 Exploiting Data Parallelism

Given an array of processor cores executing an application, the instruction fetch bandwidth determines the upper bound for packet throughput. Each processor should be kept supplied with work to keep the bandwidth utilized; at the same time, the application should be designed to avoid instruction overhead to make the best possible use of the available bandwidth. Exploiting task parallelism by dividing an application across multiple processors can convey significant benefits, but those advantages are generally tied to diminished instruction fetch efficiency. We conclude from this that in most cases applications should not be partitioned: each packet should be handled on a single processor to exploit the data parallelism of independent packets. This approach allows a simple mechanism for keeping the processor load balanced and for managing variable packet handling while making the best use of the available instruction fetch bandwidth. We can then seek other avenues for providing the potential advantages of partitioning the application to the data parallel implementation without incurring performance penalties.

Hardware Context Bottleneck A processor's hardware contexts can become a performance bottleneck if they cannot support enough concurrent contexts to keep the processor supplied with work. One of our task parallel implementations, MemRef, avoided the hardware context bottleneck by transferring packets off the processor after a remote memory reference: the receiving processor queues the packet in its local memory until the remote data had been copied in. To loosen the hardware context bottleneck within a data parallel implementation we adapted the same approach: instead of transferring the packet to another processor's local memory, we kept the packet local to the same processor by copying the context's live state to its own local memory. The data transferred to and from processors is approximately the same in both cases. In the data parallel implementation, virtual contexts decouple the number of hardware contexts from the multi-threading requirements: hardware contexts act as caches for the context state and allow multi-threading to scale beyond the available hardware contexts. The instruction block sizes and the remote memory latency determine how many virtual and hardare contexts are required to keep the processor occupied.

Instruction Fetch Bandwidth In order to increase the packet handling throughput using a fixed number of processor cycles, we must reduce the number of cycles required to execute the packet handling instructions. We can accomplish that either by expressing the packet handling work in fewer instructions or by raising the instruction fetch bandwidth. Our task parallel CPU_Config used heterogenous processor configurations to increase the bandwidth for some application paritiotns: we explored three techniques for loosening the instruction fetch bottleneck in our DataParallel implementation. The first technique, using ISA extensions to implement the packet processing work, reduced the total demand on the processor pipeline by executing fewer pipeline operations. The optimized application uses the available instruction fetch bandwidth more efficiently. The other two techniques, using dual-issue configurations to exploit ILP or issue instructions from two independent contexts, take advantage of existing processor resources to execute the same number of operations in fewer cycles, just as the streaming programming model did for the CPU_Config implementation. These techniques increase the efficiency of the hardware if parallel resources have been unused, as with our single-issue configuration on our processor cores. However, we can make a valid comparison between the execution using a dual-issue configuration on one of our processor core to the same application executing on a pair of single-issue processor cores who divide most of the processor hardware between them: each has a single instruction issue and decode path, a single ALU and four hardware contexts. The dual-issue configurations make better use of a few rarely-used shared resources such as the divide unit, but otherwise the processor efficiency does not go up. Static_ILP reduces the efficiency of the instruction fetch bandwidth by not issuing two useful instruction on every cycle: it also increases the processor and memory port contention between contexts by shrinking the instruction block sizes. Static_SMT execution is almost the same as our two single-issue processors: since it does not change the instruction blocks, the performance remains unchanged as well. The choice between Static_SMT and the simpler processors may lie in how they use external resources, such as the network and local memory.

Managing Packet Latency Packet latency is the sum of the live cycles spent executing instructions, the idle cycles spent waiting for data, and the contention cycles spent waiting for access to processor resources. Task parallelism offers a technique for reducing packet latency by executing tasks for a single packet in parallel. The latency of a packet in our application domain typically has a large fraction of its cycles waiting for data. Our experiments show that for such an application, keeping the processor occupied using multiple contexts can add a large number of contention cycles: the cumulative result is that the live cycles may be a small component of a packet's total latency.

Once the number of virtual contexts has been determined, the instruction block sizes also play a key part in determining how much contention is generated between concurrently executing contexts. The greater the delta between the smallest and largest instruction block sizes, the more contention occurs between virtual contexts. As the processor stall asymtotically approaches zero the contention, expressed as packet latency and packet jitter, grows linearly using our round-robin scheduling.

We have several options for managing packet latency for data parallel implementations. First, we could optimize the live cycles as discussed above. Reducing the live cycles targets a probably small component of the packet latency and may or may not be successful in diminishing it: depending on how the individual instruction blocks change, live cycles may be replace by contention cycles. Second, the number of virtual contexts assigned to the processor may be reduced in return for a significant drop in contention and a small increase in processor stall. Third, we can divide a large block into two smaller ones. Reducing short blocks can introduce more processor stall for a given number of virtual contexts while reducing long blocks can eliminate conflict for resources among contexts, allowing additional virtual contexts to add less packet latency. Changing the number of instruction blocks, irrespective of the related changes (either more instructions per packet or smaller instruction blocks) does not explicitly change the application performance. The benefit of dividing one large block into two is less conflict between contexts with no loss of throughput. The potential downside is higher context copy frequencies, putting more pressure on the hardware contexts and the local memory ports. Lastly, our analysis of dynamic execution showed us that executing the same instruction block for each concurrent packet sequentially grouped all the small blocks together (increasing the number of virtual contexts required) and all the large blocks together (increasing the contention contributed by each virtual context). We can greatly reduce both the virtual contexts required and the contention contributed if the packet execution is staggered.

Instruction Buffer Management The flexible packet-processing application we are targeting may tax an individual processor's instruction storage capacity and discourage executing the full application on a single processor. One benefit conveyed by any task-parallel application implementation is to reduce the number of executable footprint required by an individual processor: however, partitioning the application does not guarantee that capacity problem will be eliminated. Another option would be to change how the local memory system is used. Our Static_SMT processor configuration makes more effictive use of the local instruction store by supporting twice the bandwidth using the same instruction buffer footprint. The Static_ILP configuration does not have this advantage since it inflates the executable with 'nop' instructions where it lacks instruction level parallelism. Our two simple single-issue processors, discussed above, could conceivable share a single instruction buffer. Since instructions in packet-processing applications should exhibit predictable spatial and temporal locality, particularly at the instruction block granularity, managing instruction caches for this application domain could be a productive avenue.

6.2 Future Work

The work reported in this dissertation is based on a simplified packet processing system in order to clearly delineate the performance effects of specific system characteristics. This work can be extended by incorporating the dynamic flexibility that is our goal for our target system. Doing so would allow us not just to evaluate our conclusions in a more realistic context but also support exploration in two areas important to packet processing. First, we can address instruction cache management to prevent it from starving the processors of work. Second, we can implement packet management to keep the processor supplied with data.

Bibliography

[1]

[2]

[3] 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".

[4] Sumeet Singh Baboescu and George Varghese. Packet classification for core routers: Is there an alternative to cams? In *IEEE INFOCOM*, 2003.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. In *RFC 2475*, December 1998.

[6] Michael K. Chen, Xiao Feng Li, Ruigi 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, 2005.

[7] Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer, and Brian N. Bershad. Characterizing processor architectures for programmable network interfaces. In *Proceedings of the 2000 International Conference on Supercomputing*, 2000.

[8] Patrick Crowley, Mark A. Franklin, Haldun Hamidioglu, and Peter Z. Onufryk (ed.). *Network Processor Design, Issues and Practices Vol. 1.* Morgan Kaufman, 2003.

[9] Patrick Crowley, Mark A. Franklin, Haldun Hamidioglu, and Peter Z. Onufryk (ed.). *Network Processor Design, Issues and Practices Vol. 2.* Morgan Kaufman, 2004.

[10] Patrick Crowley, Mark A. Franklin, Haldun Hamidioglu, and Peter Z. Onufryk (ed.). *Network Processor Design, Issues and Practices Vol. 3.* Morgan Kaufman, 2005.

[11] Scott Rixner et al. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, 1998.

[12] M. Flynn. Some computer organizations and their effectiveness. In *IEEE Transactions on Computers*, volume C-91, pages 948–960, 1972.

[13] Mark A. Franklin and Seema Datar. Pipeline task scheduling on network processors. In *Workshop on Network Processors and Applications - NP3*, 2004.

[14] Christos J. Georgiou, Valentina Salapura, and Monty Dennaeu. A programmable scalable platform for next generation networking. In *2nd Workshop on Network Processors (NP2) at the 9th International Symposium on High Performance Computer Architecture (HPCA9)*, 2003.

[15] Matthias Gries, Chidamber Kulkarni, Christian Sauer, and Kurt Keutzer. Exploring trade-offs in performance and programmability of processing element topologies for network processors. In *2nd Workshop on Network Processors (NP2) at the 9th International Symposium on High Performance Computer Architecture (HPCA9)*, 2003.

[16] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *SIGCOMM*, pages 147–160, 1999.

[17] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. In *IEEE Network Special Issue*, volume 15, 2001.

[18] Pankaj Gupta and Nick McKeown. Packet classification using hierarchical cuttings. In *Proc of Hot Interconnects VII*, August 1999.

[19] J. Heinanen and Telia Finland. A single rate three-color marker. In *RFC 2697*, September 1999.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach 2nd Ed.* 1996.

[21] P. Kalapathy. Hardware-software interactions on mpact. In *IEEE Micro*, March 1997.

[22] Sridhar Lakshmanamurthy, Kin-Yip Liu, Yim Pun, Larry Huston, and Udai Naik. Network processor performance analysis methodology. In *Intel Technology Journal*, volume 6, August 15, 2002.

[23] Byeong Kil Lee and Lizy Kurian John. Npbench: A benchmark suite for control plane and data plane applications for network processors. In *IEEE International Conference on Computer Design (ICCD '03)*, 2003.

[24] Ying-Dar Lin and Yi-Neng Lin. Diffserv over network processors: Implementation and evaluation. In *Proceedings of the 10th Symposium on High Performance Interconnects (HOTI'02)*, 2002.

[25] Huan Liu. A trace driven study of packet level parallelism. In *Proc. International Conference on Communications (ICC)*, 2002.

[26] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceedings for ISCA 2000*, June 2000.

[27] Nadeem Malik, Richard Eickmeyer, and Stamatis Vasiliadis. Interlock collapsing alu for increased instruction-level parallelism. In *Micro*, 1992.

[28] B.A. (ed.) Maynard. *Honeywell 800 System.* 1964.

[29] G. Memik, W. H. Mangione-Smith, and W. Hu. Netbench: A benchmarking suite for network processors. In *Proceedings of International COnference on Computer-Aided Design (ICCAD)*, pages 39–42, November 2001.

[30] Gokhan Memik and William H. Mangione-Smith. Nepal: A framework for efficiently structuring applications for network processors. In *Second Workshop on Network Processors (NP2)*, 2002.

[31] Jathin S. Rai, Yu-Kuen Lai, and Gregory T. Byrd. Packet processing on a simd stream processor. In *Workshop on Network Processors and Applications - NP3*, February 2004.

[32] Madhusudana Seshadri and Mikko Lipasti. A case for vector network processors. In *Proceedings of the Network Processors Conference West*, pages 387–405, 2002.

[33] Niraj Shah. Understanding network processors. Master's thesis, University of California, Berkeley, September 2001.

[34] Burton Smith. The architecture of hep. In *Parallel MIMD Computation: HEP Supercomputer and its Applications, Scientific Computation Series*, pages 41–55, 1985.

[35] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principals (SOSP)*, October 2001.

[36] Dean Tullsen, Susan Eggers, and Hank Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[37] Ning Weng and Tilman Wolf. Pipelining vs. multiprocessors. In *Proceedings of Advanced Networking and Communications Hardware Workshop (ANCHOR 2004)*, 2004.

[38] Tilman Wolf and Mark A. Franklin. Design tradeoffs for embedded network processors. In *International Conference on Architecture of Computing Systems*, 2002.