Reconciling Performance and Programmability in Networking Systems*

Jayaram Mudigonda † Harrick M. Vin Stephen W. Keckler

Department of Computer Sciences, The University of Texas, Austin TX 78712 USA.
{jram|vin|skeckler}@cs.utexas.edu

ABSTRACT

Challenges in addressing the memory bottleneck have made it difficult to design a packet processing platform that simultaneously achieves both ease-of-programming and high performance. Today's commercial processors support two architectural mechanisms–namely, hardware multithreading and caching–to overcome the memory bottleneck. The configurations of these mechanisms (e.g., cache capacity, number of threads per processor core) are fixed at processor-design time. The relative effectiveness of these mechanisms, however, varies significantly with application, traffic, and system characteristics. Thus, programmers often struggle to achieve high performance from a processor that is not well-suited to a particular deployment.

To address this challenge, we first make a case for, and then develop a malleable processor architecture that facilitates the dynamic reconfiguration of cache capacity and number of threads to best-suit the needs of each deployment. We then present an algorithm that can determine the optimal thread-cache balance at run-time. The combination of these two allows us to simultaneously achieve the goals of ease-of-programming and high performance. We demonstrate that our processor outperforms a processor similar to Intel's IXP2800–a state-of-the-art commercial Network Processor–in about 89% of the deployments we consider. Further, in about 30% of the deployments our platform improves the throughput by as much as 300%.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; C.4 [**Performance of Systems**]: Design studies

General Terms: Design, Experimentation, Performance.

*This research was funded in part by the Intel Corporation and NSF ITR grant: ANI-0326001. Jayaram Mudigonda was also supported by the Intel Foundation Fellowship.

†Currently at The Hewlett-Packard Labs, Palo Alto CA USA. Email:jayaram.mudigonda@hp.com

SIGCOMM'07, August 27–31, 2007, Kyoto, Japan

Keywords: Memory bottleneck, Processor architectures, Reconfigurable architectures, Data cache, Multithreading, Packet processing, Routers.

1. INTRODUCTION

Designing the next-generation network infrastructure poses a significant challenge: *develop a packet processing platform that simultaneously achieves the two goals: ease-of-programming and high performance.* This platform facilitates: (1) rapid development and deployment of novel (and potentially disruptive) network architectures, services, and protocols; and (2) experimentation on realistic scales. Not surprisingly, such a platform forms the basic building block for the next-generation commercial routers [15], as well as research testbeds [2].

A key challenge in achieving these twin-goals of ease-of-programming and high performance is the *memory bottleneck.* This bottleneck—caused by the ever-widening gap in performance between processors and memory—has been a major source of concern for all of computing. Unfortunately, this problem is further aggravated in networking systems for two reasons. First, over the past decade, the link bandwidths, and hence the rate of packet arrivals at a network system, have increased significantly. Second, packet processing applications are becoming increasingly sophisticated; for each packet, they make hundreds of accesses to large data structures that often cannot be fit in the on-chip memories. In most modern packet processing applications (e.g., network monitoring, intrusion detection, virus scanning, and protocol conversion), the time to process a packet is dominated by memory access overhead. Thus, the problem of achieving high packet throughput often reduces to the problem of dealing with the memory bottleneck effectively.

Existing research on addressing the memory bottleneck in networking systems can be broadly classified into three categories: (1) algorithms that reduce the number of memory accesses by exploiting specific characteristics of the traffic or control data (see [35] for examples of optimized route lookup schemes); (2) techniques for efficiently utilizing the exiting general-purpose hardware (e.g., exploiting the DRAM row-locality for fast packet buffers as proposed in [19]); and (3) special-purpose dedicated hardware mechanisms for specific applications (e.g., the route and classification caches proposed in [12] and [38]). Unlike all these proposals, in this paper, we consider widely-applicable processor-based mechanisms for dealing with the memory bottleneck. We believe that, with the processor architectures undergoing a radical design shift (from complex single core to multiple simpler

cores), now is an opportune time for the networking community to put across its wishes to processor architects.

Today's processors include two primary mechanisms to address the memory bottleneck: *multithreading* and *data caching*. These mechanisms have complementary strengths. Whereas multithreading exploits the inherent *packet-level parallelism* to hide memory access latencies (reduce processor stalls resulting from memory accesses), caches exploit *data locality* to reduce the memory access overhead [25].

The relative effectiveness of these two mechanisms is influenced by several factors. For instance, applications that update shared state for each packet serialize the execution of packets; hence, such applications favor processors with larger data caches but with smaller number of cores and threads. Similarly, systems with limited off-chip bandwidth also prefer larger caches. In contrast, systems with large off-chip memory bandwidth but long latencies, as well as those that are engineered for the worst-case traffic favor multithreading (since caches do not improve the worst-case). Consequently, the ideal combination of caching and threading required to maximize packet throughput varies significantly across *deployments* (defined as combinations of application, traffic, and system characteristics).

Current processors, however, provide only fixed configurations of cache capacity and number of threads, as the chip area, a common resource required by both caching and multithreading, must be statically allocated at chip-design time. Hence, programmers are often left with the task of achieving high performance on a processor that is not well-suited to the deployment characteristics. For instance, on processors with inadequate support for multithreading, programmers are required to develop efficient application-specific techniques to exploit parallelism, as the overhead of generic software threads can be prohibitively expensive. On the other hand, on processors with small or no caches, programmers often have to resort to tricky optimizations to reduce the number of memory accesses. For instance, careful orchestration of thread scheduling is required to prevent multiple threads from fetching the same address repeatedly [21]. Thus, platforms based on today's processors, achieve high performance often at the expense of ease-of-programming.

In this paper, we ask two fundamental questions. In order to reconcile performance and programmability in a wide-variety of deployments: (1) what architectural mechanisms should a processor support? and (2) how should one exercise these mechanisms? We make the following four contributions.

1. Case for Malleability: In Section 3, we make a case for a *malleable* processor core that allows the run-time tradeoff between cache capacity and number of threads. We observe that, it is plausible to build such a processor because both caching and multithreading need the same abstract resource, namely fast memory close to the processor core; while caching uses it to hold data expected to be accessed soon, multithreading uses it hold thread-specific registers. We compare this malleable processor to a processor similar to Intel's IXP2800 because, IXP2800 is a state-of-the-art commercial multicore multithreaded processor designed for networking workloads. We show that, the malleable processor outperforms the IXP2800-like processor in as many as 89% of the deployments. Further, in 30% of the deployments, the malleable processor achieves a throughput gain of 300% when compared to the IXP2800-like processor, and 60% when compared to the optimal fixed architecture that maximizes the packet throughput across all deployments.

2. Realizing the Malleability: In Section 4, we address the challenge of realizing such a malleable processor. We base our architecture on the novel concept of a *malleable storage* block that can be dynamically partitioned to hold registers as well as cached data. Our malleable storage design consists of a small multiported register cache (MRC) and a primary data cache; the data cache holds thread-specific registers in pinned lines as well as normal cached data. With this architecture, configuring different amounts of threading and caching simply involves changing the amount of data cache reserved for pinning thread-specific registers. We design a novel register access predictor to efficiently bridge the bandwidth and latency gap between MRC and data cache.

3. Exercising the Malleability: In Section 5, we consider the problem of automatically exercising the malleable processor core. We develop a low-overhead run-time adaptation algorithm that automatically finds and maintains the optimal thread-cache balance. We exploit the key intuition that processor utilization (and hence the packet throughput) first increases and then decreases with the increase in number of simultaneous threads. The packet processing platform created by combining this scheme with our malleable processor, requires only thread-safe code and programmers are freed, to a great extent, from the tedious and mostly ad hoc performance tuning exercise.

4. Experimental Evaluation: Finally, in Section 6 we demonstrate that our platform, besides simplifying the programmability, also achieves high performance by showing that it performs within 8% of an ideal malleable processor, which suffers none of the practical reconfiguration overheads that are incurred by our processor.

The rest of the paper is organized as follows. Section 2, describes our experimental methodology. We provide a detailed presentation of our four contributions in Sections 3 to 6. We then discuss our findings in Section 7, and the related work in Section 8. Finally, we present our conclusions and opportunities for further work in Section 9.

2. EXPERIMENTAL METHODOLOGY

In this section, we first describe the set of deployments we consider. Informally, we define a deployment as the tuple <`application`, `packet trace`, `off-chip memory latency`, `memory BW`>. We then present the details of the processor core simulator. Finally, we explain the area model and tools we use to estimate the area requirements of various architectural structures.

2.1 Deployment Scenarios

Packet Processing Applications: Functions of a packet processing system can be broadly classified into: (1) *data-plane* that processes the regular packets and (2) *control-plane* that involves management tasks such as keeping the route table updated. In this paper, we focus on data-plane applications because, in most systems they process the vast majority of packets and consume most of the processing resources. Table 1 shows the applications we consider; these applications cover each of the following data-plane functions:

(1) verify the integrity of incoming packets; (2) classify them into flows; (3) process the packets, and (4) schedule the outgoing packets. Our application set includes most of the two popular benchmark-suites for packet processing [23, 36], as well as the full data-plane functionality specified in the NP forum [5] benchmarking guidelines. Networking systems composed from these applications often include some additional code for purposes such as passing the packet descriptors from one application to another and managing the list of free buffers. We do not include this code in our simulations because, it can be reasonably assumed that, in most cases such glue code requires only a small fraction of additional processing and memory accesses, and hence does not change any of the qualitative conclusions of this research.

Packet Traces and Control Data: We experiment with packet traces collected at a spectrum of locations along the Internet hierarchy: ANL, FRG, MRA (from NLANR [26]), and UNC (from the University of North Carolina). While ANL trace (access link between Argonne National Lab and its ISP) represents the traffic at the network edge, MRA (link between two large networks: Merit and Abilene) represents the traffic at the network core. UNC, FRG (Front Range Giga PoP) represent intermediate locations, with UNC being closer to the edge. Each of these traces contain between 0.5 and 5.0 million packets and our experiments reach steady state after about 75,000 packets. The qualitative conclusions remain same across all these traces. Since the ANL and MRA traces often lie at the extremes in their requirements for caching and multithreading, we only present the results for these two traces.

For the route lookup applications, we construct the route table using the data published by the *RouteViews* project [8]. However, IP addresses found in most traces do not match any of these real prefixes since all publicly available traces are anonymized to preserve privacy. We *de-anonymize* them by substituting every occurrence of an IP address with another randomly selected address for which the route table contains a prefix. This process, not only preserves traffic patterns but also conforms to the traffic generation guidelines recommended by the Network Processor Forum [5].

For the `vscan` application, which scans the packet payload for virus signatures, we use the signature database published by *Snort* [7]. Unfortunately, publicly available traces (that are long enough for our experiments) do not include packet payload. For our experiments, we consider two scenarios: (1) random payload; and (2) payload containing popular web pages. By doing so, we characterize the *common case* where the packet contents match a virus signature only rarely.

System Model and Parameters: We consider a single-chip, homogeneous multicore processor where the individual cores are multithreaded. We assume coarse-grain multithreading that switches threads on an off-chip memory access; coarse-grain threading has been shown to achieve most of the performance benefits of much more complex fine-grain simultaneous multithreading (SMT) processors [13]. Each core contains a local data cache that is shared by all of its threads; the cores share a single off-chip memory level. We use the two important parameters to characterize the off-chip memory: (1) the access *latency* and (2) the available *bandwidth*

To study realistic deployment scenarios, we consider multiple values of latency and bandwidth—the choice is guided by the values of these parameters supported in modern network processors. For instance, the Intel IXP2800 Network Processor supports multiple levels of memory with access latencies of 60, 150, and 300 cycles [21]. Further, it supports a cumulative SRAM bandwidth of 0.64 references per CPU cycle. Hence, for our experiments, we consider several different latency values (50, 100, 200, 300, 400, and 500 cycles); and memory access bandwidths (ranging from 0.04 to 2.56 refs/cycle).

2.2 Simulation Tools

A complete survey of the space of deployment scenarios requires us to quantify the effectiveness of a large number (100s) of processor core configurations over an even larger number (1000s) of deployments. Each of these simulations must process millions of packets and requires more than a day to finish on today's PCs. Hence, full execution based simulation in all cases quickly becomes infeasible. To make this study possible without significantly sacrificing the accuracy, we split the simulation into two phases. First, we adapt the Simplescalar [6] instruction emulator to record an execution trace for a given application and packet trace. Then, we reuse this execution trace to drive a multithreaded processor core simulator configured for different number of threads and amount of cache.

Trace Generation: We use the `sim-safe` simulator from the Simplescalar toolkit, which simulates a simple RISC instruction set. We enhanced `sim-safe` to generate an *execution trace* that is partitioned into *blocks*, where each block represents the processing of a packet. Each block consists of (1) the arrival time of the packet, taken from the packet trace, and (2) the sequence of all the ALU instructions, mutex operations, and memory accesses performed. In this trace, we do not record accesses to globals and call stack because, for all the applications we consider, these data are small enough (around 100 bytes) to fit into registers and fast on-chip memories [21].

Trace Execution on a Multithreaded Core: Our multithreaded core simulator models a very simple in order five-stage pipeline with a full bypass network. The basic parameters, and the range of values they are set to, are shown in Figure 1(a). This simulator processes the execution trace and emulates the behavior of multiple threads interleaved over a single pipeline. The packet arrival time stored in each of the execution trace blocks is used to match the arrival time of packets from the network line interface. Simulator tracks time at the granularity of a clock cycle. We model all the major sources of wasted processor cycles such as the lock contention, queueing within the memory subsystem, and the thread switch overhead. Thread switches occur when the current thread makes a memory access or is blocked on a mutex, and incur a minimum latency of 2 cycles [21]. Our simulator collects a wide-variety of statistics such as various types of misses, distribution of thread run-lengths, and memory queue lengths.

We validated our simulator by comparing the processor and cache performance it reports to those of Simplescalar's `sim-out-order` configured with similar parameters . In addition, we spot-checked the memory queueing times reported by our simulator by comparing them to those of a cycle accurate simulator of IXP2800 [3].

Functionality	Application	Source	Notes
Integrity verification	`checksum`	Free BSD	Protects headers in TCP, UDP and IP packets (RFC-1071) [4]
	`md5`	R.S.A Inc.	MessageDigest 5 (MD5). Mostly used to protect the payload (RFC-1321) [4]
Classification	`classify`	UT-Austin	Hashes the five-tuple: ⟨`srcIP,dstIP,srcPort,dstPort,protocol`⟩ [18].
Route Lookup (Longest prefix match)	`patricia`	Free BSD	Patricia tree. Can handle non-contiguous masks. Used in many end-systems [31].
	`bitmap`	UT-Austin	Employs bitmaps to compress trie nodes. Used in many commercial routers [16].
	`bsol`	UT-Austin	Binary search using hash tables. Has the best known avg. comp. complexity [35].
	`ixp`	IXA SDK 3.0	Designed for IXP series of NPs. Two tries are searched simultaneously [3].
Metering (Prioritize packets)	`srtcm`	IXA SDK 3.0	Enforces a *single* mean rate and a peak burst. (RFC-2697) [4].
	`trtcm`	IXA SDK 3.0	Enforces *two* independent rates: mean and peak. (RFC-2698) [4].
	`tswtcm`	UT-Austin	Enforces mean and peak rates over sliding windows. (RFC-2859) [4].
Header processing	`stream`	Snort 2.0	TCP receive-side processing. Reassembles byte streams out of packets [7].
	`portscan`	Snort 2.0	Detects portscan attack if too many ports are accessed too quickly [7].
Payload processing	`cast`	SSLeay Lib	Encryption scheme. Used in Virtual Private Networks (VPNs) (RFC-2612) [4]
	`vscan`	Snort 2.0	Pattern matcher. Scans packet payload for virus signatures [7, 37].
Scheduler	`drr`	UT-Austin	Deficit Round Robin. Found in many commercial routers [35].

Table 1: Applications (UT-Austin = We developed)

2.3 Chip Area Estimation

For the purpose of estimating chip area, we model a single multithreaded core as consisting of a *thread-specific*, a *thread-independent*, and a *data cache* region. The thread-specific region includes all the architectural components that must be scaled with number of threads, such as register files, status words, and program counter. We assume that the size of the thread-specific region grows linearly with number of threads supported by the processor core. The thread-independent region includes such components as one or more functional units, pipeline logic, and instruction fetch and decode units. The data cache region contains the data cache.

We derive the area estimates for the thread-specific, and the thread-independent regions based on the IXP2800 design (see Figure 1). IXP2800 uses 0.13 micron technology; it consists of 16 processing cores (referred to as microengines) and an XScale core; we do not consider the Xscale core in our area calculations because it is used only for the less-frequent control-plane tasks. The 16 microengines occupy approximately 29% of the 14.12×18.88 mm^2 die, for a total of $4.83mm^2$ per microengine. 27% (1.3 mm^2) of the area of each microengine constitutes the thread-specific region. Since each microengine supports 8 threads, each thread occupies $0.163mm^2$. We refer to this area as one *thread-equivalent* (*th-eq* for short) and use it as a unit of allocation for our analysis. Note that for IXP2800, the thread-independent area of each microengine is about 21 thread-equivalents, while the total area occupied by the 16 microengines is approximately 475 thread-equivalents.

We estimate data cache size using Cacti-3.2 [1] with a 0.13 micron fabrication process. With these parameters, a single thread-equivalent chip area can accommodate about 128 2-way associative 128-bit lines. Once more than 8 thread-equivalent chip area is allocated to data cache, every additional thread-equivalent chip area results in the addition of about 256 cache lines.

Finally, because packet processing workloads are inherently concurrent, and we assume the processors and their workloads are homogeneous, we estimate that the chip area and the memory bandwidth requirements grow linearly with number of processors. This assumption is fair because, packet processing systems take care to balance the load across all the processor cores. Hence, we simulate only one processor core and extrapolate to the performance of multiple cores.

3. CASE FOR MALLEABILITY

In this section, we first show that the optimal combination of multithreading and caching that maximizes the packet throughput varies significantly with application, packet traffic, and memory subsystem characteristics. We then demonstrate that a malleable architecture that can adapt to these variations can achieve impressive throughput improvements compared to the current fixed architectures.

3.1 Influence of Deployment Characteristics

We consider a single processor core with 16 th-eqs of chip area available to be split between threads and cache. For all the experiments in this subsection, we fix the context switch overhead to 2 cycles and the off-chip memory bandwidth to 0.64 references/cycle, and unless otherwise noted, off-chip memory access time to 150 cycles. These choices are guided by the characteristics of typical IXP2800-based systems [21].

Application Characteristics: Figure 2(a) depicts, for each application driven with the ANL trace, the optimal balance between number of threads and cache capacity that maximizes the packet throughput sustained by the processor core. Within each bar, the darker and the lighter regions, respectively, represent the number of th-eqs assigned to threads and data cache. It can be seen that the balance varies over almost the entire range.

On the one extreme, applications : `cast`, `checksum`, and `md5`, use up most of the space ($\geq$ 15) for threads. This is because these applications scan the payload and thus do not benefit from cache. On the other extreme lie applications `drr`, `stream`, and `portscan`. In these applications, a significant fraction of the per-packet processing is not parallelizable due to read-write data that must be accessed in mutual exclusion. Hence, these applications do not benefit from large number of threads and much of the available chip area (12 of the 16 th-eqs) is allocated to the data cache.

Many other applications–`ixp`, `patricia`, `bitmap`, `trtcm`, `srtcm`, `tswtcm`, and `classify`–fall in the middle. These applications exhibit considerable locality for data accesses and are also amenable to parallelization. Hence, the available chip area is split somewhat evenly between threads and cache (with about 5-8 th-eqs allocated for threads). Thus, we conclude that the applications differ from each other widely in terms of the characteristics that influence the balance between multithreading and caching.

Pipeline	5-stage in order
Num. threads	Configurable (1 to 64)
Regs per thread	64
Switch latency	2 cycles
L1 cache latency	Configurable (1 to 4 Cycles)
L1 cache associativity	2,4-way
L1 cache line-width	16 bytes
L1 cache capacity	Configurable (0.25 KB to 64 KB)
Memory Latency	Configurable (50 to 500 cycles)
Memory BW	Configurable (0.04 to 2.54 refs/cycle)
Memory Bus-width	Configurable (32 bits)

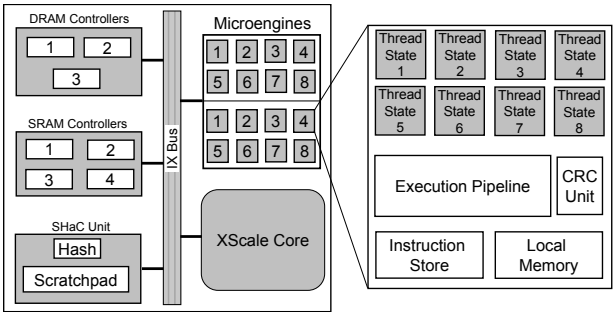
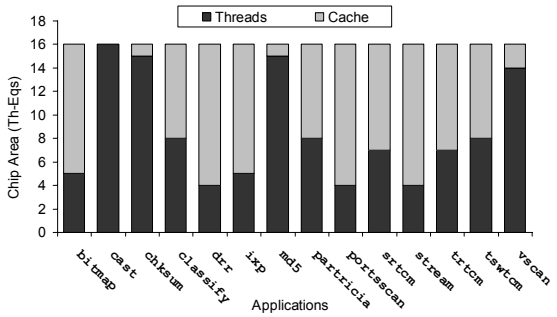
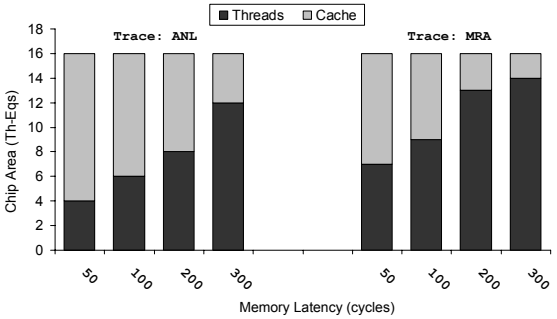


Figure 1: Simulation Parameters and the Intel IXP2800 Processor



(a) Application Characteristics



(b) Traffic and Memory Subsystem Characteristics

Figure 2: Effect of Deployment Characteristics on the Thread-cache Balance

Memory Subsystem Characteristics: Figure 2(b) shows, for the application `classify`, the change in balance between threading and caching with increasing memory access latency for both the ANL and MRA traces. It illustrates that, for both these traces, the balance shifts from a cache-heavy configuration (e.g., with 4 threads and 12 thread-equivalents allocated to cache) at 50 cycle latency to a thread-heavy configuration (e.g., with 12 threads and 4 th-eqs allocated to cache) at 300 cycle memory latency. Restricting the off-chip memory bandwidth influences the balance similarly; for brevity, we omit these results.

Packet Traffic Characteristics: Figure 2(b) shows that the optimal thread-cache balance for `classify` application, when driven with MRA trace, shifts towards multithreading. This shift happens because of two reasons. First, the data locality exhibited by `classify` depends upon how packets from different flows (as identified by different fields in packet header) are interleaved. For instance, the frequency of re-occurrences of the packets belonging to the same flow determines how often a hash bucket is accessed. Second, compared to ANL, MRA trace is collected from a link closer to the network core and hence, contains traffic aggregated from a larger number of sources. Thus, in MRA trace, packets of a given flow are separated by a larger number of unrelated packets (of other flows). This results in larger working sets and diminished effectiveness of data caches.

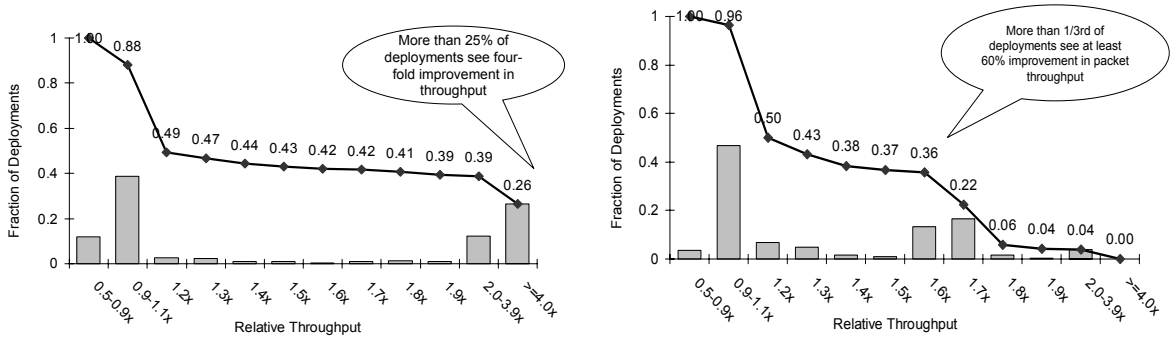
For most of the remaining applications – `ixp, patricia`, `bitmap`, `trtcm`, `srtcm`, and `tswtcm` – balance changes substantially from ANL to MRA. Further, all these applications favor a larger number of threads with traces closer to the network core (i.e., traces containing larger number of flows) [24]. On the other hand, the thread-cache balance does not change for payload processing applications such as `cast`, `md5`, and `vscan`, because data accesses in these applications are not determined by the header field values, and hence are not influenced by trace characteristics [24].

3.2 Benefits of Malleability

We now demonstrate that an architecture that can adapt to the wide variations in thread-cache balance required by a spectrum of deployments can achieve significant throughput gains. We consider a *malleable* architecture consisting of a fixed number of cores, but allows, within each core, the cache capacity to be traded off for the number of threads to best-suit the deployment requirements. We observe that, it is plausible to build such a processor because both caching and multithreading need the same abstract resource, namely fast memory close to the processor core; while caching uses it to hold data expected to be accessed soon, multithreading uses it hold thread-specific registers.

We call a malleable architecture *ideal* if it can match, in all possible configurations (i.e., all combinations of threads-per-core and cache-per-core), the performance of the fixed architecture with the exact same configuration. That is, the ideal malleable architecture does not sacrifice performance to implement the thread-cache tradeoff.

We compare the throughput supported by such an ideal malleable processor with that of two fixed architectures with the same chip area: (1) Intel's IXP2800 (with 16 cores, 8 threads per core, no data cache), and (2) an optimal fixed configuration (6 cores, 7 threads and 45 th-eqs of data cache per core). We consider the ideal malleable processor with the optimal configuration of 9 cores and 26 thread-equivalents



(a) Compared to the IXP2800 (b) Compared to the Optimal-fixed

Figure 3: Throughput Improvements with Malleability

of space per core that can be configured to support any desired combination of threads-per-core and cache-per-core. In [24], we show that these optimal fixed and optimal malleable configurations maximize the packet throughput across all deployments for a given chip area of IXP2800.

Figures 3(a) and (b), respectively, show, over all deployments (described in Section 2.1), the throughput improvements the ideal malleable processor can achieve as compared to the IXP2800 and the optimal fixed architecture. The bars in each figure show the fraction of the deployments in which the ideal malleable processor achieves a given performance improvement. The curve above the bars shows, for a given performance improvement of x, the percentage of the deployments where the malleable processor achieves an improvement of *at least* x. The figures show that when compared to IXP2800, the malleable processor improves the throughput by at least 100% in 38% of the deployments, and by at least 300% in as many as 25% of the deployments. Even when compared to the optimal fixed configuration, the malleable processor improves performance by at least 60% in 35% of the deployments, and by at least 70% in about 25% of the deployments. In a small fraction of deployments (4% and 12% when compared to the optimal-fixed and the IXP2800 respectively), however, malleability leads to loss in throughput. This is because, the optimal malleable configuration includes fewer (9) cores compared to the IXP2800 (with 16 cores), and a smaller cache (26 thread-equivalents) compared to the optimal-fixed (with 45 thread-equivalents of cache). Thus, the malleable processor suffers in deployments at the extremes of parallelism-locality tradeoff, which can benefit from the extra cores and cache capacity available in the IXP2800 and the optimal-fixed respectively.

4. REALIZING MALLEABILITY

The main requirement in realizing such a malleable processor is a block of malleable memory that can hold two fundamentally different types of data: thread-specific registers that need fast accesses at large bandwidths, and regular cached data that need large capacity (see table 2). However, a single storage structure cannot simultaneously meet all these requirements as the limitations of circuit-level designs force a sacrifice of one of latency, bandwidth, and capacity for the remaining two. Hence, any realization of a malleable memory must consist of two levels: a small, fast, high-bandwidth memory and a large, slower, low-bandwidth memory. Thus, the challenge of building a malleable memory is to make the fast memory appear to have the capacity of the slower memory. We observe that this in principle, is the same problem addressed in the traditional memory hierarchy. Hence, in this section, we first consider two traditional solutions for managing memory hierarchies, namely caching and double-buffering, and show that they are not effective in achieving malleability. We then derive the principle of predictive register prefetching and show that it can be easily implemented with little overheads.

	Registers	Data-$	Malleable Store
Latency (cycles)	Low (1)	Moderate (2-8)	Low (1)
BW (refs/cycle)	High ($\bar{3}$)	Low (0.25-0.33)	High (up to 3)
Capacity per thread	Small (0.25 KB)	Large (few KBs)	Large (few KBs)
Addressing	Index	Associative	Associative

Table 2: Malleable Storage: Required Features

The high-level architecture of our malleable processor core is shown in Figure 5. As explained above, it consists of a two storage structures. The multiported register cache (MRC) provides the high-bandwidth low-latency register accesses to the pipeline, and the data cache acts as a backing store for the MRC and hence holds, along with the regular cached data, thread-specific registers in pinned lines. We refer to the set of data cache lines that holds registers as *context-partition* and the remaining lines as *data-partition*. In this architecture, the effective capacity of the malleable memory is determined by the data cache capacity because, the MRC only holds *copies* of some of the data cache contents,namely subset of registers for a few threads. Thus, to support the full range of thread-cache tradeoff, we must treat the chip area devoted to the MRC as an overhead and minimize it.

4.1 Register Caching and Double-buffering

We now show that two well-known solutions to managing memory levels, namely LRU caching and double-buffering require very large MRCs to be effective. The MRC implementations for these techniques are shown in Figure 4. With register caching, when the register needed is missed in the MRC, the pipeline stalls until the register is fetched from the context-partition of the data cache. To minimize these stalls register caching must provision enough capacity in the MRC. On the other hand, register double-buffering eliminates these stalls by using two register files: an *execution-*

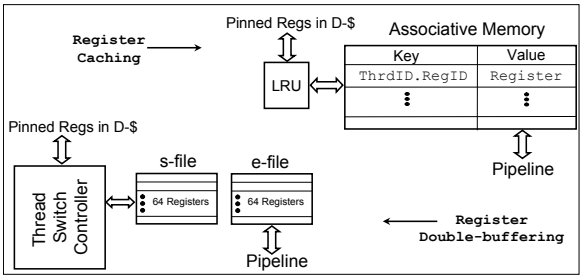


Figure 4: Candidate MRC Architectures

file (e-file) and a *swap-file (s-file)*. While the *e-file* is serving the register accesses of the current running thread, *s-file* is filled with the registers for the next thread to run. When the current thread makes an off-chip memory access, a context switch happens and the *e-file* and *s-file* are swapped. If needed, the swap is delayed, and the pipeline is stalled, until all the registers (of the next thread) are fetched into the *s-file*. Thus, to ensure a good performance of double-buffering, adequate bandwidth must be provisioned between the MRC and the data cache.

For these two techniques, we evaluate the chip area requirements of the MRC, considering a processor core with 26 thread-equivalents of available area (for the MRC, and data cache) [24]. To perform within 10% of the ideal malleable processor (defined and studied in Section 3), register caching requires at least 1264 lines of fully-associative cache. Using Cacti we determined that, to support such a cache, the MRC requires 12 thread-equivalents (out of the 26 available). This large cache size can be explained by three factors: (1) the very high hit rates required because of the expensive misses (pipeline stalls), (2) poor locality in register accesses [11], and (3) the linear growth in working set size with number of threads. On the other hand, double-buffering requires the MRC to support a bandwidth of 16 refs/cycle, and the necessary banks, ports, and associated wiring require about 14 thread-equivalents of space. This exorbitant bandwidth requirement can be explained by the short run-lengths of these memory intensive applications. Thus, register caching and double-buffering do not utilize the chip area efficiently enough to support the full range of malleability. A comprehensive study of these, and several other recently proposed register caching techniques can be found in [24].

4.2 Predictive Register Prefetching

Our malleable architecture combines the advantages of register double-buffering and caching. While double-buffering completely eliminates stalls due to missing registers in the MRC, it does so by using enormous bandwidth to prefetch *all* the registers, whether they are used or not. On the other hand, caching minimizes the bandwidth usage. However, it waits too long to fetch a necessary register—until the pipeline actually stalls. Thus, in principle, to combine the strengths of both these approaches, one should prefetch only those registers that are needed by the next thread. In our architecture, we instantiate this principle with the help of a novel register access predictor.

Predicting Register Accesses: Our malleable processor architecture, shown in Figure 5, consists of two register files and a predictive prefetcher. The execution-file (*e-file*) holds the registers for the current executing thread, while the swap-file (*s-file*) is used to prefetch the registers of the next thread that are predicted to be used. Both these files can hold all 64 registers of a single thread. Each of the registers is augmented, for the use of the prefetcher, with three management bits: *valid* (`V`), *read* (`R`), and *modified* (`M`). In addition, the program counter (PC) where execution resumed and the thread identifier are captured in the *s-file* and *e-file*.

Our prefetching scheme is based on the key insight that the PC of the next thread is a strong predictor of the registers that need to be loaded from the backing store into the *s-file*. This insight is based on two observations. First, there is a good correlation between the PC and the control flow (the instructions sequence); this correlation is what makes branch prediction possible. Second, since registers are hard coded in the instructions, a given sequence of instructions always produces the same sequence of register accesses.

Our predictor captures this correlation with the help of a per-thread table that maps a given PC to a bit mask of registers that would be accessed when the thread begins to execute from that PC. We store these tables in the context-partition of data cache along with the per-thread registers. Later in this section, we demonstrate that the data cache has enough capacity and bandwidth for these tables. In what follows, we explain the predictive prefetching in detail. We use the terms `crnt`, `prev`, and `next` to denote, respectively, the thread that is currently running on the pipeline, the thread that ran immediately before `crnt` and the thread that is going to run immediately after `crnt`.

Execution: When `crnt` writes to a register, it sets the corresponding `V` and `M` bits in the MRC; no fetch from the backing store is done. When `crnt` reads an invalid register (with the `V` bit unset), the pipeline stalls until the missing register is fetched from the context-partition of the data cache. When the value returns, it is saved in *e-file* with the `V` and `R` bits set. The `R` bit signifies that a register was read before it was written during the thread's execution, and hence should be prefetched the next time the thread resumes at the same PC that started the current run. Registers that are written before they are read need not be prefetched– because the value gets overwritten anyway. Thus `R` bit is not set for such registers (i.e., ones with the `M` bit set).

Thread Switch: On a thread switch, the two register files *s-file* and *e-file* are logically swapped. At this instant, `crnt` becomes `prev`, and `next` becomes `crnt`. The prefetcher first copies back all of `prev`'s modified registers from the *s-file* into the context-partition. It then updates `prev`'s predictor table. The new predictor table entry is the 64-bit mask constructed from all of the `R` bits set by `prev`. The entry to modify is located by indexing the PC that launched `prev`'s latest run. Then the prefetcher determines the set of registers to prefetch for `next`, by indexing `next`'s predictor table with the PC where the execution is going to resume. It then proceeds to prefetch these registers into *s-file* while the current thread runs.

Accuracy of Our Predictor: To evaluate the accuracy, we simulate the prefetching scheme for all applications under all deployments. We measure the *miss rate*–defined as the fraction of register references that resulted in data cache accesses, because the predictor failed to anticipate their usage.

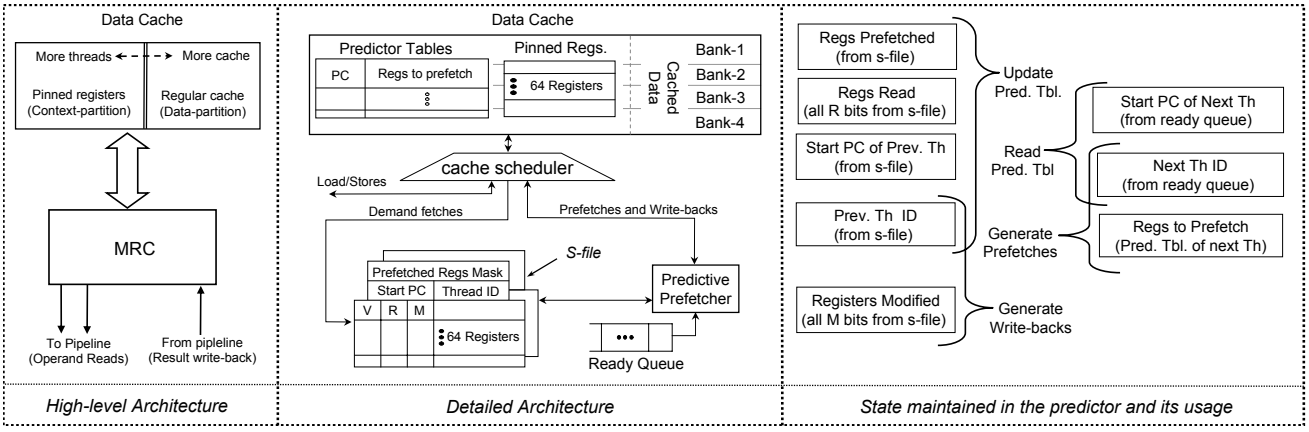


Figure 5: Our Malleable Core Architecture.

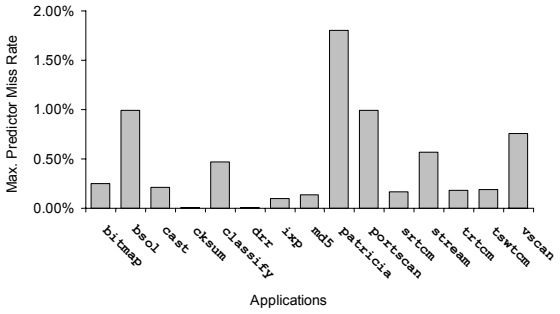


Figure 6: Miss rate Achieved by Our Predictor

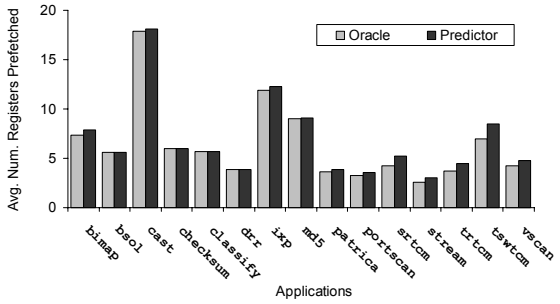


Figure 7: Accuracy: Our Predictor vs. Oracle.

Figure 6 shows the maximum miss rate observed over all the deployments. Figure 7 compares, for each application, the average number of registers prefetched by our predictor (the darker bars) to that of an oracle with the perfect knowledge of the next thread's register accesses (lighter bars). First, the miss rate suffered by our scheme is very small ($< 2\%$). That is, our scheme successfully prefetches a vast majority of the registers that are required. Second, the number of registers fetched by our scheme is very close to the that of the oracle. These two observations put together demonstrate that, our scheme prefetches a register *if and only if* it is required by the next thread. This conclusion validates our intuition that PC is a good predictor of the register accesses to follow. Not surprisingly, the worst-suffering application is `patricia`, which has one of the most complex control flows.

Predictor Overheads: To perform at its best, our predictive prefetching requires: (1) enough fast storage for the per-thread predictor tables and (2) adequate bandwidth between the cache and the MRC to ensure that the prefetching completes before the current thread stalls on a memory access.

We carried out a detailed analysis of the maximum table size required for all the applications, and over all the deployments [24]. We found that for most applications the table size rarely exceeds 32 entries, with `stream` requiring the largest table with just 185 entries. We explain these small table sizes based on two observations. First, packet processing applications execute only a small portion of the code under normal conditions; bulk of the code addresses error conditions that rarely happen. Second, cache misses are typically correlated with only a subset of the memory access instructions. Consequently, even within the code for normal processing, only a small subset of PCs account for the vast majority of cache misses and hence thread switches.

We exploit their small sizes and store the per-thread predictor tables in the primary data cache. Our analysis shows that, for a processor with a modest number of threads, the data cache has enough spare capacity for these tables. For instance, consider the 26 th-eqs of area available in our malleable processor core. This area, after accounting for the needs of the MRC, translates to about 2KB for each of the 26 threads. This is substantially more than the 1736 bytes (256 bytes for the 64 32-bit registers and 1480 bytes for the 185 64-bit predictor entries) required by a thread of the most demanding application, namely `stream`. Then, we measure the maximum bandwidths required between the MRC and the data cache, taking into account the accesses to the predictor table along with the prefetching related transfers. We found that the bandwidth required is about 1.08 refs/cycle, which can be easily provisioned in practice.

5. EXERCISING MALLEABILITY

Having designed the necessary architectural mechanisms, we now present a simple run-time adaptation scheme that automatically selects and maintains the optimal thread-cache balance. The design of our scheme is guided by the following three observations

First, selecting the optimal thread-cache combination at deployment time based on programmer-driven profiling suffers several drawbacks. Deployment-time profiling requires programmers to specify several time-varying and hard-to-estimate parameters such as traffic patterns. Further, often the resulting systems: (1) are not robust to traffic fluctuations and (2) are hard to maintain and upgrade.

Second, in each deployment, $U(n)$, the processor utilization (and thus the packet throughput) with n active threads, first increases with n, and after reaching a peak, begins to decrease, to never rise again. First, increasing the threads from n to $n+1$ increases the performance because, the aggregate working set of $n+1$ threads fits in the cache and the additional thread only helps to hide the memory latency further. However, since the aggregate working set size is a non-decreasing function of the number of threads, at some value of $n+1$ the aggregate working set no longer fits in the cache. At this stage the utilization begins to fall for three reasons: (1) cache miss rate goes up because, not only has the working set grown larger, but also the available cache capacity has been reduced since the new thread claims cache lines for its registers and predictor entries; (2) higher miss rate combined with larger number of threads leads to increased contention for off-chip memory bandwidth, eventually making the system bandwidth limited; and (3) higher miss rate means more thread switches and more wasted cycles. We observe that these arguments hold for most system settings and applications.

Third, in our architecture, increasing the number of threads is fairly straightforward. We simply start the new thread and the necessary data cache lines for its registers and predictor entries get automatically allocated and pinned on demand. On the other hand, terminating a thread and releasing its data cache lines requires the cache lines for *all* the threads to be unpinned because, we cannot determine all the lines held by a particular thread's predictor table entries. This unpinning effectively releases only the lines held by the terminated thread because, the remaining threads quickly re-pin most of their lines as they continue to execute. Thus, terminating a thread leads to short period with a slight ($\approx 6\%$) increase in the per-packet processing time. This degraded performance, however, lasts only for a few hundred packets.

The first of the these observations requires the adaptation scheme to be sufficiently light-weight to allow run-time usage. Given the second observation about the shape of $U(n)$, a simple linear or binary search through the space of all possible active threads yields the optimal thread-cache balance. The third observation dictates that, to minimize the run-time overhead, the algorithm should avoid reducing the number of threads as much as possible.

Our algorithm (Algorithm 1) does a simple linear search, but reduces the number of times a thread is terminated by using asymmetric step sizes; the number of threads is incremented by one while searching upwards, but is decremented by two while searching downwards.

In spite of such linear search, once the optimal thread-cache balance is found at the system startup, our scheme tracks the changes in the balance with only a small number (about 2-4) of adjustments. Further, during each adjustment, it makes a very small number (≈ 6) of memory accesses and executes less than 40 instructions. Thus, our scheme imposes little overhead during normal operation and can be run on the data path itself with practically no performance loss (see Section 6).

Algorithm 1 Adapt Thread-Cache Balance

Notation:
N_{max}: Max number of threads
N_c: Current number of threads
$\lambda[n]$: Throughput with n threads

```
1: for n = N_c + 1 to N_max do
2:    λ[n] = trialRun(n)
3:    if (λ[n] < λ[n − 1]) then
4:       break
5:    end if
6: end for
7: if ((n − 1) > N_c) then
8:    return (n − 1) /* increasing works */
9: end if
10: /* increasing does not work */
11: for n = N_c − 2 down-to 1 with step (-2) do
12:    λ[n] = trialRun(n)
13:    if (λ[n] < λ[n + 2]) then
14:       break
15:    end if
16: end for
17: /*also try the last thread skipped*/
18: λ[n + 1] = trialRun(n + 1)
19: return (λ[n + 1] > λ[n + 2])?(n + 1) : (n + 2);
```

6. EXPERIMENTAL EVALUATION

In this section we quantify the effectiveness of our platform. First, we explain the parameters of our experimental setup. We then present the results that compare, for all the deployment scenarios, the performance of our architecture to the ideal malleable processor and the state-of-the-art IXP2800.

6.1 Experimental Setup

Processor and Memory: The processor and memory system models correspond to the parameters described in Section 2. Deployments multiplex up to 26 threads over the pipeline. Thread switches happen on data cache misses and take a minimum of 2 cycles. We assume an instruction store within each core, and thus simulate a perfect instruction fetch that never stalls the pipeline.

MRC: The MRC consists of the two register files described in Section 4. We provision both these register files with two read/write ports and one exclusive write port (three ports total), resulting in a total write bandwidth of 3 refs/cycle. Although, as noted in Section 4.2, the desired average bandwidth is about 1.08 refs/cycle, these extra ports are required to accommodate the occasional burstiness in the accesses between the MRC and the data cache and thus to minimize the wasted cycles between thread switches. However, these extra ports make the MRC about 3% slower compared to the traditional register files that only have one write and two read ports. We take this slowdown into account in our performance evaluation [24].

Data cache: We use a 4-banked, 2-way set-associative cache with 16-byte lines. The MRC described above requires approximately 2 thread equivalents of space which leaves 24 thread equivalents to the data cache, corresponding to 54

KB. This data cache is about 75% slower than the MRC, resulting in an access time of 2 cycles. However, we assume that the data cache is pipelined and thus accepts a new request every cycle.

Cache bank scheduler: The cache scheduler implements a simple fixed priority. Requests from the memory stage of the pipeline have the highest priority. Demand misses from the operand read stage have the second highest priority. All the prefetch related requests and the fills from L2 cache are served at the lowest priority.

Run-time adaptation: Adaption occurs every 100,000 packets and each trial lasts for 7,500 packets. If the packet stream stops during a trial, adaptation terminates unsuccessfully. When the packet stream starts again, the processor resumes execution with the same number of threads as before the unsuccessful adaptation. We run the adaptation algorithm on each processor core that performs data path processing, because overhead imposed by the adaptation algorithm is negligible (about 6 memory accesses and 40 instructions every 7,500 packets).

Traffic and control data: We utilize the the traffic traces, route tables described in Section 2. These execution traces do not contain accesses to the stack. However, we capture the worst-case effect of these accesses by modeling a bank conflict, on each cycle, with a cache bank that has at least one prefetch request pending. Thus, we believe that our experiments constitute a pessimistic scenario for our platform.

6.2 Evaluation Results

Comparison with Ideal Malleable Platform: We compare the packet throughput of our platform to that of the ideal malleable processor defined and studied in Section 3. Recall that, ideal malleable processor achieves malleability with no implementation overheads; it matches, in all possible thread-cache configurations, the performance of the equivalent fixed architecture that supports the same number of threads and same amount of cache, but includes separate register files for each thread. Further, the ideal platform has the oracular knowledge of the optimal thread cache balance for each deployment. In contrast, our processor suffers the following overheads: (1) longer cycle times due to the slower MRC (compared to traditional register file), (2) stalls that may happen due to inaccuracies in prefetching, and (3) run-time adaption and the potential utilization loss when the adaptation is not perfect. For this experiment, both processors include 9 cores each with 26 thread-equivalents of space for threads and cache. We set the context switch overhead and cache access latency to 2 cycles each. The remaining parameters, namely application, off-chip memory latency and BW are determined by the deployment.

Figure 8(a) depicts, the relative throughput of our architecture with respect to the ideal malleable processor. For a given relative throughput of x, the bars in each figure show the fraction of deployments that experience an improvement of x, where as the line above the bars shows the fraction of deployments that see an improvement of *at least* x. We make the observation that our architecture performs, in as many as 92% of the deployments, within 6% of the ideal malleable processor. Further, the maximum loss of 8% happens only in less than 1% of the deployments.

Comparison with IXP2800 and Optimal-fixed: In Section 3 we quantified the throughput improvements of ideal malleable platform over IXP2800 and the fixed processor that optimally partitions the available area. Our architecture achieves virtually the same performance improvements because, our architecture performs within 8% of the ideal malleable processor. Hence, the plots that compare our architecture to the IXP2800 and the optimal-fixed are identical to those shown in Figure 3; we omit these for space reasons. We conclude that, our platform improves the throughput, in as many as 30% the deployments, by 300% compared to the IXP2800 and by 60% compared to the optimal-fixed.

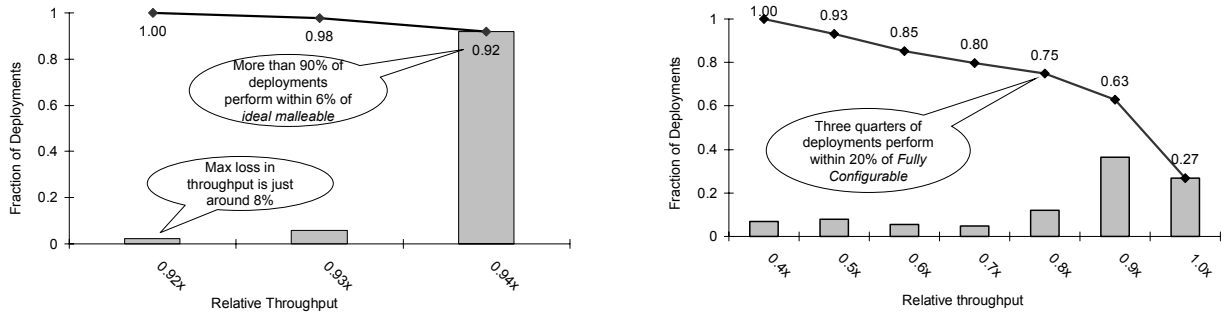
Comparison with Hypothetical Fully Configurable Processor: To measure how effectively our architecture utilizes a given chip area, we compare our processor to a *hypothetical fully configurable* processor, that allows each deployment to select not only number of threads and cache capacity, but number of cores as well. Figure 8(b) shows, for a given chip area of IXP2800, the relative performance of our malleable processor with respect to this hypothetical processor. Our architecture performs within 10% of this fully configurable processor in 63% of the deployments and within 20% in 75% of the deployments. Thus we conclude that our architecture not only achieves significant gains over the state of the art, but also realizes most of the performance of the much more capable hypothetical processor.

7. DISCUSSION

Need for Malleability: We examine three plausible arguments against malleable processors. First, consider a heterogeneous processor that includes both cache-heavy and thread-heavy cores. On such a processor, one may achieve high throughput by carefully mapping each application to the best-suited cores. However, we observe that programming such heterogeneous systems for high performance remains at least as hard as today's state of the art, while our malleable system is substantially easier to program than the state of the art.

Second, one could build a small number of chips each optimized for only a subset of deployments with a common characteristic. For instance, separate chips could be designed for the network edge and core. We considered several such classifications of the deployment space. We found that, in most cases, the ideal thread-cache balance varies even within a subset. The single malleable core designed for the entire set of deployments outperforms, or performs comparably, to the fixed processors optimized for each subset [24].

Third, for systems that are engineered to support a given throughput do not require malleable processors because, such systems do not benefit from throughput improvements the run-time adaptation achieves. We argue that even these systems benefit from the simplified programmability offered by malleable processors. Programmers can focus on issues other than memory bottleneck, and the thread-cache balance can be fixed during the profiling phase that most of these systems undergo to ensure the throughput. Further, malleable processors hold a significant cost advantage. Larger volumes associated with a single malleable chip, that can be used in a wide-variety of systems, allows the chip manufacturer to better amortize the significant costs associated with chip design and validation, there by bringing down the price.



(a) Compared to Ideal Malleable Processor (b) Compared to the Hypothetical Fully Configurable Processor

Figure 8: Effectiveness of Our Malleable Packet Processing Platform

Simplifying the Architecture: We observe that, for each context switch, only a few ($\approx$ 5) registers are needed to be prefetched. If, only a few "hot" registers such as stack pointer account for most of the prefetches, a compiler or profiler could provide hints to the hardware and thus eliminate the predictor. However, we found that the set of 5 most frequently prefetched registers (by oracle) differs significantly across applications, and in all applications, account for only about 40-60% of the prefetches required for stall-free operation. Consequently, prefetching only these registers leads to too many demand-fetches and poor performance (about 65% of our predictive prefetching scheme.)

8. RELATED WORK

Existing work on memory bottleneck in networking systems can be classified into three main categories: (1) algorithms for reducing off-chip memory accesses; (2) schemes that efficiently utilize general-purpose hardware; and (3) special-purpose hardware.

Algorithmic Techniques: Most work in this area focuses on specific problems with the goal of reducing the number of memory accesses. For instance, the route lookup scheme proposed in [16] reduces memory accesses by using bitmaps to compress the nodes of the lookup trie; a good overview of other such schemes can be found in [35]. Similarly, techniques for memory access reduction are proposed in [22, 33] for large hash table implementations, and in [17] for online statistics collection.

Exploiting General-purpose Hardware: Partridge et al. describe how to best-exploit the memory hierarchy of an Alpha processor to achieve the lookup rates needed for their 50Gbits/Sec router [27]. Bjorkman et al. examine the effectiveness of utilizing shared memory multiprocessor systems for parallel protocol processing [10]. Hasan et al examine methods for better exploiting DRAM row-locality specifically for packet headers and payload [19].

Special-purpose Hardware Support: A wide and highly pipelined memory architecture for a collection of processor cores is proposed in [30]. A Memory hierarchy designed for a specific lookup scheme is presented in [9]. Special memory buffers for packet headers and payload are described in [20]. Special-purpose hardware caches for route lookup and Layer-4 classification are proposed in [12] and [38] respectively. The ability of SMT, CMP, and superscalar architectures to exploit instruction level parallelism in networking applications is studied in [13, 14]. Several problems in designing high-speed network interfaces for end-systems are addressed in [32].

General-purpose Systems: Register caching has been utilized to build large yet fast register files for SMT and wide-issue processors in [11, 28]. The general technique of register double-buffering has been used to hide synchronization latencies in classical multiprocessor systems in [34]. The TRIPS processor includes a memory block that can act as either a cache or a simple physical memory [29].

9. CONCLUSIONS AND FUTURE WORK

A packet processing platform that can achieve high packet throughput and is easy to program forms an important building block for the next-generation commercial networks as well as research testbeds. In this paper, we address the memory bottleneck–a key challenge in designing such a platform. We make four contributions: (1) we make a case for a malleable processor core that supports the dynamic tradeoff between cache capacity and number of threads; (2) we then design a novel malleable architecture that facilitates this tradeoff; (3) we present an adaptation algorithm that automatically finds and maintains the optimal thread-cache balance at run-time, there by simplifying the programmability; and (4) we demonstrate that our processor outperforms a processor similar to IXP2800–a recent NP–in about 89% of the deployments we consider, and in about 30% of the deployments it quadruples the packet throughput.

Future Work: Although this paper focuses on packet processing systems, we believe that many of our ideas and insights are applicable more broadly. For instance, an interesting line of work examines the fundamental locality-parallelism tradeoff in the broader class of high-throughput systems such as web and database servers. Another avenue investigates how to utilize our insight that register access can be accurately predicted to build very large yet fast register files for wide-issue general-purpose processors. Finally, apart from the memory bottleneck, there exist other programmability challenges, such as making it easy to program multithreaded processors, that require further research.

10. ACKNOWLEDGMENTS

We thank Prof. Mike Dahlin of UT Austin, Dr. Raj Yavatkar, Eric Johnson, Aaron Kunze of Intel Corporation for their constant help throughout this research. We also thank Prof. Patrick Crowley of Washington University, St. Louis, for his comments on an earlier draft of this paper.

11. REFERENCES

[1] CACTI3.2 http://tinyurl.com/yqu8a5.

[2] Global Environment for Network Innovations. http://www.geni.net.

[3] Intel IXA SDK 3.0. http://tinyurl.com/2ltuwu.

[4] Internet RFC Archive. http://www.faqs.org/rfcs/.

[5] NPF Benchmarks; http://tinyurl.com/2rbz6g.

[6] Simplescalar 3.0. http://www.simplescalar.com/.

[7] Snort IDS; http://www.snort.org.

[8] University of Oregon Route Views Project. http://www.routeviews.org/.

[9] J.-L. Baer, D. Low, P. Crowley, and N. Sidhwaney. Memory hierarchy design for a multiprocessor look-up engine. In *Proc. of the IEEE Conf. on Parallel Architectures and Compilation Techniques*, pages 206–216, 2003.

[10] M. Bjorkman and P. Gunningberg. Locking Effects in Multiprocessor Implementations of Protocols. In *Proc. of the ACM SIGCOMM Conf. on Communications*, pages 74–83, 1993.

[11] J. A. Butts and G. S. Sohi. Use-based register caching with decoupled indexing. In *Proc. of the ACM Intl. Symp. on Computer Architecture*, pages 302–313, 2004.

[12] T.-C. Chiueh and P. Pradhan. Cache Memory Design for Network Processors. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture*, pages 409–418, 2000.

[13] P. Crowley, M. E. Fiuczynski, and J.-L. Baer. Characterizing Processor Architectures for Programmable Network Interfaces. In *Proc. of the Intl. Conf. on Super Computing*, pages 54–65, 2000.

[14] P. Crowley, M. E. Fiuczynski, and J.-L. Baer. On the Performance of Multithreaded Architectures for Network Processors. Technical Report TR2000-10-01, University of Washington, 2000.

[15] W. Eatherton. "The Push of the Network Processing to the Top of the Pyramid", Keynote at the Symp. on Architectures for Networking and Communications Systems, 2005.

[16] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *ACM SIGCOMM Computer Communication Review*, 34(2):97–122, 2004.

[17] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proc. of the Internet Measurement Workshop*, pages 75–80, 2001.

[18] P. Gupta and N. McKeown. Algorithms for Packet Classification. In *IEEE Network - The Magazine of Global Internetworking*, pages 24–32, March 2001.

[19] J. Hasan, S. Chandra, and T. N. Vijaykumar. Efficient Use of Memory Bandwidth to Improve Network Processor Throughput. In *Proc. of the ACM Intl. Symp. on Computer Architecture*, pages 300–313, 2003.

[20] S. Iyer, R. R. Kompella, and N. McKeown. Analysis of Memory Architecture for Fast Packet Buffers. In *Proc. of the IEEE Workshop on High Performance Switching and Routing*, pages 368–373, 2001.

[21] E. J. Johnson and A. Kunze. *IXP 2800 Programming*. Intel Press, 2003.

[22] S. Kumar and P. Crowley. Segmented hash: an efficient hash table implementation for high performance networking subsys. In *Proc. of the Symp. on Architectures for Networking and Communications Systems*, pages 91–103, 2005.

[23] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: A Benchmarking Suite for Network Processors. In *Proc. of the IEEE/ACM Conf. on Computer-Aided Design*, pages 39–42, 2001.

[24] J. Mudigonda. *Addressing the Memory Bottleneck in Packet Processing Systems*. PhD thesis, Universtity of Texas at Austin, 2005.

[25] J. Mudigonda, H. M. Vin, and R. Yavatkar. Managing Memory Access Latency in Packet Processing. In *Proc. of the ACM Conf. on Measurement and Modeling of Computer Systems*, pages 396–397, 2005.

[26] NLANR Network Traffic Packet Header Traces. http://pma.nlanr.net/Traces/.

[27] C. Partridge et al. A 50-Gb/s IP router. *IEEE/ACM Transactions on Networking*, 6(3):237–248, 1998.

[28] M. Postiff, D. Greene, S. Raasch, and T. Mudge. Integrating superscalar processor components to implement register caching. In *Proc. of the Intl. Conf. on Super Computing*, pages 348–357, 2001.

[29] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proc. of the ACM Intl. Symp. on Computer Architecture*, pages 422–433, 2003.

[30] T. Sherwood, G. Varghese, and B. Calder. A Pipelined Memory Architecture for High Throughput Network Processors. In *Proc. of the ACM Intl. Symp. on Computer Architecture*, pages 288–299, 2003.

[31] K. Sklower. A Tree-Based Packet Routing Table for Berkeley Unix. In *Proc. of the Winter USENIX Conference*, pages 93–103, 1991.

[32] J. Smith, E. Cooper, B. Davie, I. Leslie, Y. Ofek, and R. Watson, editors. *IEEE Journal on Selected Areas in Communications (JSAC)*. IEEE, Feb 1993.

[33] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *Proc. of the ACM SIGCOMM Conf. on Communications*, pages 181–192, 2005.

[34] V. Soundararajan. Dribble-back registers: A technique for latency tolerance in multiprocessors. Technical Report Technical Memo-474, LCS, MIT, 1992.

[35] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. M. Kaufmann, 2004.

[36] T. Wolf and M. Franklin. CommBench-A Telecommunications Benchmark for Network Processors. In *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software*, pages 154–162, 2000.

[37] S. Wu and U. Manber. A Fast Algorithm for Multi-pattern Searching. Technical Report TR-94-17, University of Arizona, 1994.

[38] J. Xu, M. Singhal, and J. Degroat. A Novel Cache Architecture to Support Layer-Four Packet Classification at Memory Access Speeds. In *Proc. of the IEEE Conf. on Computer Communications*, pages 1445–1454, 2000.