

On the Performance of Multithreaded Architectures for Network Processors

Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer*

Technical Report 2000-10-01
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195

{pcrowley,mef,baer}@cs.washington.edu

Abstract

With the ever-increasing performance and flexibility requirements seen in today's networks, we have seen the development of programmable network processors. Network processors are used both in the middle of the network, at nodes composing the backbone of the Internet, as well as at the edges of the network in enterprise class routers, switches, and host network interfaces. The application workloads for these processors require significant processing capacity but also exhibit packet-level parallelism.

In this paper, we assess the performance of processor architectures well suited to take advantage of packet-level parallelism, namely, a simultaneous multithreading processor (SMT) and chip-multiprocessors (CMP). First, in the spirit of keeping these devices as general-purpose as possible, we show that packet classification can be performed in software with overheads to deliver packets to the network processor of the order of 10% as compared to an ideal hardware implementation. Second, we simulate the execution of three network usage scenarios requiring different amounts of processing power. We find that both architectures perform well and that the choice of architecture might depend on whether one wants to stress hardware (CMP) or software (SMT) simplicity.

* This work was supported in part by NSF Grant MIP-9700970 and a gift from PMC-Sierra.

1. Introduction

Computer networks have assumed a preeminent role in providing essential communication in business, education, and daily life. As networking technology advances, it is often the case that signaling data across wires or fibers is no longer the limiting factor in network performance. Indeed, it is the management of data in the network -- whether at the edge to alleviate the load on the server, or within the network for packet routing and processing-- that presents the performance bottleneck. As a result, vendors are currently building network interfaces around powerful microprocessors. A market for these *network processors* has appeared, and a number of products are available today using a wide range of processor architecture technologies. Despite the growing importance of this application, there is a scant amount of published data allowing an assessment of the suitability of various architectures. The goal of this study is to address this situation in the case of processors that allow concurrent execution of multiple threads, namely chip-multiprocessors (CMP) and simultaneous-multithreading (SMT).

In this paper, we quantitatively evaluate the effectiveness of these two architectures when used as network processors. Our previous work has shown that as a result of packet-level parallelism, CMP and SMT can provide high performance when executing network applications, e.g., IP route lookups and MD5 authentication, in isolation. In this work, we simulate the execution of a more realistic, multi-programmed workload. We first investigate the performance tradeoffs involved with performing packet classification, or message demultiplexing, in software versus hardware. Then, we simulate the operation of a programmable network interface, complete with operating system, on a suite of three network scenarios that require a varying degree of processing power from the network processor.

In our performance evaluation, we use a simulator that is cycle accurate with respect to instruction issue and execution, cache accesses, memory bandwidth and latency, as well as to memory contention between the processor and DMA transfers caused by network send and receive operations. As a result, we are able to accurately simulate and measure the performance of the aforementioned processor architectures in the context of a programmable network interface.

The contributions of this study are two-fold. First, we consider the efficacy of software-based packet classification and compare its performance to that of ideal packet classifying hardware. Second, we quantitatively compare the

performance of SMT and two CMP processors when executing a suite of three multi-programmed network workloads.

The remainder of this paper has the following organization. Section 2 presents background information on programmable network interfaces (PNIs) and previous work. Our experimental apparatus, including system design, workloads, architectures and OS characterization, is discussed in Section 3. Section 4 considers tradeoffs between hardware and software-based packet classification. Results from the simulation of our multi-programmed workloads are presented and analyzed in Section 5. The paper ends with conclusions in Section 6.

2. Background

There is no standard definition for “a network processor” or a “programmable network interface”. In this paper, we consider network processors that can be used both in the middle of the network, at nodes composing the backbone of the Internet, as well as at the edges of the network in enterprise class routers, switches, and host network interfaces. Thus, we do not consider network interfaces that are directly attached to the memory bus [15] or more elaborate, but less general-purpose, communication co-processors for MPP’s such as the Magic chip used in the Stanford Flash multiprocessor [13] or the Meiko CS-2 network interface [4]. Even with this restriction, the range of functionality of network processors is quite large. Until recently, most network processors were “communication processors” whose application workload consisted principally of simple packet forwarding and filtering algorithms based on the addresses found in layer-2 or layer-3 protocol packet headers. The Myrinet Lanai processor [6] is such a device. Today, however, the role of network processors is expanding from simple packet forwarders to general-purpose computing/communications systems processing packets at layer-3 and above [18]. The application workloads include traffic shaping, network firewalls, network address and protocol translations (NAT), and high-level data transcoding (e.g., to convert a data stream going from a high-speed link to a low-speed link). Additionally, with the tremendous growth and popularity of the Web, many network equipment manufacturers are touting devices that can transparently load balance HTTP client requests over a set of WWW servers to increase service availability [1]. Table 1 further describes a representative set of these tasks.

Many of these emerging applications require significant processing capacity per network connection. Furthermore, the processing requirements within these domains are unique in that, generally speaking, performance must be sustained at a level equivalent to the speed of the network itself. While the clock speed of high-performance

processors is increasing every year, the rate of increase is not as fast as that of the data rate in the middle of the network; hence the need to look at enhanced processor architectures. Fortunately, workloads for network processors have an inherent characteristic: network packets or messages, which are the basic unit of work for these applications, are often independent and may be processed concurrently. This packet-level parallelism can be exploited at the architectural level to achieve the sustained high-performance demanded by fast networks. Accordingly, network processor parts, such as Intel's IXP1200 [14], IBM's Rainier [12], and those from various startup companies, such as Sitera's PRISM IQ2000 family [20], use parallel architectures, such as multiple processor cores on a single chip, to match this packet-level parallelism.

Our previous work [7] has confirmed that the key to scalable high performance, while executing network applications in isolation, such as those used in IP forwarding or VPN IP security, is to exploit the parallelism available between packets. In particular, we found that these applications did not have enough instruction-level parallelism (ILP) to achieve good performance on either a single sophisticated out-of-order, superscalar processor or a fine-grain multithreaded processor. The performance of both of these processors, in particular the number of instructions issued per cycle, depends on the ILP available in a single thread of execution. However, thread-level parallelism, as supported on SMT and CMP, can be used to achieve good performance by issuing instructions simultaneously from multiple threads. While these results have led us to narrow our architectural focus to processors that support concurrent thread-level parallelism, the workloads that we investigated lacked two important components. First, there was no need for packet classification since a single program handled all packets, and,

Tasks	Description
Packet Classification/Filtering	Claim/forward/drop decisions, statistics gathering, and firewalling.
IP Packet Forwarding	Forward IP packets based on routing information.
Network Address Translation (NAT)	Translate between globally routable and private IP packets. Useful for IP masquerading, virtual web server, etc.
Flow management	Traffic shaping within the network to reduce congestion and enforce bandwidth allocation.
TCP/IP	Offload TCP/IP processing from Internet/Web servers to the network interface.
Web Switching [1]	Web load balancing and proxy cache monitoring.
Virtual Private Network IP Security (IPSec)	Encryption (3DES) and Authentication (MD5)
Data Transcoding [8]	Converting a multimedia data stream from one format to another within the network.
Duplicate Data Suppression[15]	Reduce superfluous duplicate data transmission over high cost links.

Table 1. Representative tasks and application specific packet processing routines.

second, the network processor did not exercise a full system workload. In this work, we intend to address each of these issues.

Packet classification refers to how messages are demultiplexed. When a packet is received from the network, the packet headers must be examined in order to determine what should be done with it. Web requests, for example, are identified as TCP packets destined for port 80. A packet classifier would take a packet, or packet header, as input and return as a result the target application. Much work has been done to design packet classifiers in both hardware [2, 11] and software [5, 8, 21]. Packet classification in network devices has often been implemented in custom silicon in the interests of high-performance. However, the state-of-the-art today in software based packet classification provides good performance and greater flexibility than ASIC solutions. In this study, we investigate the performance tradeoffs between hardware and software based classification. We use Engler’s publicly available DPF packet filter[8], which uses dynamic code generation to create highly optimized software packet filters. While no longer the fastest software implementation described in the literature[5, 21], DPF has some nice properties including performance that is insensitive to the number of filters and good portability. We compare the performance of this software implementation to the performance of an ideal hardware implementation that classifies packets instantaneously. We believe this to be a fair comparison since any particular system can be engineered such that the hardware classifier is not the bottleneck.

Our previous work shows that when applications are executed independently, SMT and CMP are able to achieve performance that is linear in the number of simultaneous threads of execution supported by the hardware. However, one would not expect any of these applications, in isolation, to tax the resources of either architecture. Our goal, in simulating more realistic workloads where several applications are executing simultaneously, is to assess the performance and scalability of the SMT and CMP architectures.

3. Experiment Setup

This section describes the experimental setup used in this study to investigate the operation of a PNI under realistic system workloads. This experimental setup involves: the overall system design of the PNI, the set of programs that process packets, the CPU architectures used on the PNI, and the baseline performance of the operating system that governs the operation of the device.

3.1 Overall System Design

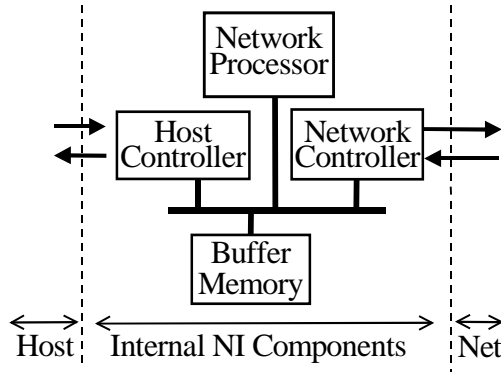


Figure 1. Generic Programmable Network Interface Architecture

We first consider the overall design of the system. As shown in Figure 1, a generic PNI consists of host and network controllers, a memory, a CPU, and a bus interconnect. The execution environment for such a device can be described as having a store-and-forward design. The *store* and *forward* stages simply transfer message data into and out of the NI's buffer memory. The *process* stage invokes application-specific handlers. We illustrate this design by considering the various operations involved for a network processor receiving a packet from the network, performing some function on the packet, and forwarding the packet to some host (cf. **Figure 2**).

As messages arrive, a high-speed network interface using DMA delivers the message into the network processor buffer memory. From a functional viewpoint, this arrival process can be modeled as a FIFO queue of messages awaiting handling by the processor. Since our main interest is in the processor architecture, we do not dwell on the type of network interface (sonet vs. atm vs. gigabit Ethernet etc.) but we do model bus contention between the transfer of messages in memory and access to memory by the processor(s). Furthermore, we assume that the DRAM (or embedded DRAM) that we use can provide the bandwidth requirements of the processor and network

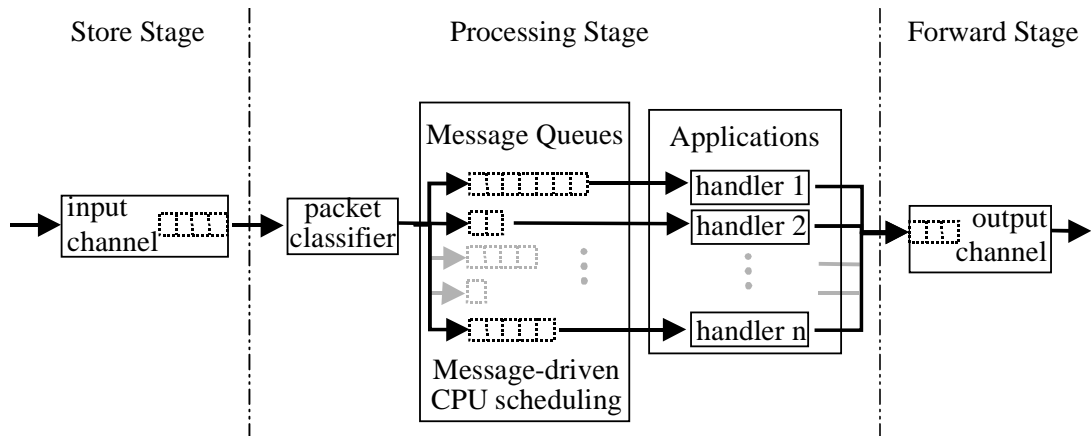


Figure 2. The Store-Process-Forward stages of messages flowing through a programmable NI.

Application	Insts Executed per Message	Loads/Stores (%)	Ctrl Flow (%)	Other (%)
IP-FORWARD	~200	25.4	12.7	61.9
HTTPMON	~400	35.0	12.1	52.9
MD5	~2000	10.7	2.8	86.5
3DES	~40000	17.8	1.2	81.0

Table 2. Benchmark characteristics.

controllers. Messages are then classified (cf. Section 4) and dispatched to some application-specific handler function to be processed. Finally, the messages may then be forwarded on to an output channel. To achieve high throughput and low latency it is important that messages be pipelined through these stages. Thus, the goal is to sustain three forms of concurrent operation: message reception from the input channel, message processing on the NI, and message transmission to the output channel. It is the task of the system software to schedule these concurrent operations such that each pipeline stage is occupied.

Varying degrees of programmability may be used to implement each of these stages. Our previous work compared both hardware and software implementations of the store stage. In this study, the network processor operates under a multi-programming load that requires a more sophisticated OS interaction, so we consider only the software implementation. Also in our previous study, as mentioned earlier, we assumed that all packet classification was performed in dedicated hardware and that it was never the performance bottleneck in the system. The first set of experiments discussed in this paper revisits this assumption and considers both hardware and software implementations. Simulation of the forward stage is difficult, since the network interface could reside, for example, either on the I/O bus of an end-host or on a fast interconnect with many other line cards in a general-purpose router [18]. From our perspective, which is the operation and performance of a single node, the creation or forwarding of packets will only increase demand on resources outside the node such as the busses connecting processor, memory, and controllers. We will consider these effects in future work as our workloads and system organization push the limits of our bus interconnect.

We have designed and implemented an execution environment for a PNI with the above-mentioned properties for both uniprocessor and multiprocessor configurations. Fiuczynski et al. [9] describes this execution environment in more detail, and the relevant issues are discussed in Section 3.4.

3.2 Workloads

We have previously identified a suite of tasks for programmable network interfaces. From this list, shown in Table 1, we have selected three tasks for use in this study. Using these, we have designed three multi-programmed usage scenarios to investigate the operation of a programmable network interface in three different points in the network (cf. Section 5.1 for more details). Each task we consider is implemented in terms of the applications listed in Table 2. In each scenario, one application is called most often but there are also calls to the other three. We consider: (1) a wide-area network (WAN) IP router, which primarily forwards packets, (2) a web switch that monitors HTTP requests and connections, and, lastly, (3) a virtual private network (VPN) node that decrypts and authenticates most traffic to the node. Each of these scenarios takes place in a different level in the network and demands different computational resources, depending on the dynamic mix of applications that comprise the workload. To drive our simulations, we use packet traces of real traffic collected on our local network and by researchers at Harvard. Each scenario has a different packet classification policy, which maps packets to applications, and hence each scenario executes the applications in different frequencies. We assume that there is no waiting time between packets, i.e., the input queue of **Figure 2** is never empty.

Table 2 gives a basic characterization of each application used in these scenarios. The first benchmark, IP-FORWARD, performs address-based packet lookups in a tree data structure and is a component of conventional layer-3 switches and routers[17]. In test runs of IP-FORWARD we perform lookups into a routing table of 1000 entries, a size representative of a large corporate (or campus) network. HTTPMON monitors the protocol state of TCP-based web connections. This connection state can be used, for example, to distribute requests intelligently amongst nodes in a web server farm. MD5 is a message digest application used to uniquely identify messages. MD5 computes a unique signature over the data in each packet and is used for authentication. Finally, 3DES is an encryption routine that is used here to encrypt the full payload of a packet.

An examination of the instruction and data cache performance for IP-FORWARD, HTTPMON, MD5, and 3DES indicates that these applications are compute bound as both the instruction and data working sets fit within L1 caches of size 32K and greater (miss rates generally less than 1%; most misses are compulsory). Each application executes a different number of instructions per packet. Moreover, the compute loads are increasing: IP-FORWARD executes the fewest instructions progressing up to 3DES, which executes the most, as indicated in Table 2.

Instruction type	Latency
ALU	1
Multiply	8 & 16
Divide	16
Branch	2
Load	1
Store	1
Synchronization	4

Table 3. Instruction latencies.

3.3 Architectures

As mentioned in Section 2, the opportunity for processing packets in parallel and the lack of ILP in the type of applications that we are interested in has led us to focus our investigations on architectures that support the concurrent execution of multiple threads and to not consider any further sophisticated out-of-order superscalar or fine-grained multithreaded processors since the performances of the latter on the workload of interest are quite inferior [7]. In this section, we briefly describe the three high-performance processors used in this study. Their instruction set is an extension of the Alpha ISA with support for byte and 16-bit word load/store operations. The functional units in each processor are capable of executing each of the instruction types found in Table 3 with the latencies indicated in the table. Table 4 details the specific characteristics of the three processors and their memory systems.

Chip-Multiprocessor (CMP). A CMP partitions chip resources rigidly in the form of multiple processors [16]. As an example, a 4 processor CMP has 4 separate register files, 4 separate fetch units, etc. CMP has the benefit of permitting multiple threads to execute completely in parallel. However, CMP has the drawback of restricting the amount of total chip resources a given thread may utilize to those resources found within its local processor. In the experiments described in this study, we consider two different CMP processors: one has a single functional unit per core (and therefore will exploit no ILP) and is referred to as CMP, and the other (CMP2) has two functional units per core. Each processor core has a private L1 (both data and instruction), while a single L2 is shared amongst all processors. The L1 caches are kept coherent using an invalidation protocol similar to the one described in [16]. In some of our experiments, we will vary the number of processors in CMP from 1 to 8 and in CMP2 from 1 to 4, i.e., our basis of comparison is the number of functional units.

Simultaneous Multithreaded (SMT). An SMT architecture [22] has hardware support for multiple thread contexts and extends instruction fetch and issue logic to allow instructions to be fetched and issued from multiple threads each cycle. As a result, overall instruction throughput can be increased according to the amount of ILP available *within* each thread and the amount of thread-level parallelism available *between* threads. The SMT architectures we simulate in this study fetch up to eight instructions each cycle from a maximum of two threads, and the number of functional units vary from 1 to 8.

In the simulations that follow, we have clocked all processors at 500 MHz. We are aware that the experiments are therefore biased in favor of SMT since the simple core processors of CMP and CMP2 could be much faster (as much as 25% for CMP as projected in a recent study [3]).

3.4 Baseline SPINE Performance

In this study, we use the SPINE operating system [9], which implements an execution environment for the network processor described in Section 3.1. SPINE devotes a single thread to pull messages off of a Myrinet network controller. To achieve low-latency messaging this OS thread is pinned to a single processor context that polls the

Property	CMP 8 processors, each 1-way	CMP2 4 processors, each 2-way	SMT 8-way 8 thread contexts
# of CPUS	8	4	1
Total issue width	8	8	8
Total fetch width	8	8	8
# of architectural registers	32*8	32*4	32
# of physical registers	32*8	32*4	100
# of integer FUs	8	8	8
BTB entries	256	256	256
Return stack size	12*8	12*4	12
Instruction queue size	8*8	8*4	32
I cache (size/assoc/banks)	32K/2/8	32K/2/8	32K/2/8
D cache (size/assoc/banks)	32K/2/8	32K/2/8	32K/2/8
L1 hit time (cycles)	1	1	1
Shared L2 cache (size/assoc/banks)	512K/1/1	512K/1/1	512K/1/1
L2 hit time (cycles)	10	10	10
Memory latency (cycles)	68	68	68
Thread fetch width	8	4	2
Cache line Size (bytes)	64	64	64
L1-L2 bus width (bytes)	32	32	32
Memory bus width (bytes)	16	16	16

Table 4. Architectural details of the maximum configurations for each architecture. Cache write policies: L1 write-through, no write-allocate, L2 write-back.

network controller for incoming packets, as interrupt handling would be too costly. Incoming packets are deposited into a global queue, and a work-list based parallel computation model is used to schedule work (i.e., unprocessed packets) across the “application processor contexts.” Each of these application processors runs a single SPINE worker thread that waits for available work items, consumes it from the work-list, processes it, and then waits for the next work item, and so on.

This asymmetric division of work (i.e., OS thread and application threads) may be detrimental to overall performance in the CMP case, as idle resources on separate processor contexts cannot be used towards the OS thread. In contrast, the dynamic allocation of processor resources automatically performed in hardware by SMT hides this asymmetric division rather well. However, a more realistic implementation of SPINE for CMP would divide the work among the processors in a more symmetric fashion to distribute the load more evenly across each CMP processor context, but this is left for further study. In discussing performance results in this section and in forthcoming ones, we will take this static allocation factor (as well as our comments on clocking at the end of the last paragraph) into consideration.

The performance metric that we use in this paper is the *number of IP packets processed per second*. Since our focus is on the network processor architecture, we assume an unbounded line rate so that we can assess the range of speeds the network processor can support. Three factors will then be of primary importance: processor clock rate, I/O management, and processor architecture. As mentioned earlier, the processor clock rate in our simulations is fixed at 500 MHz (our previous work [7] investigates this parameter). The techniques for I/O management in our simulated system are a consequence of our choice of Myrinet[6] as a network interface and SPINE as an operating system. Processor architecture is varied first in the forms of SMT, CMP and CMP2. A second parametric variation is the number of functional units present in each architecture. In the case of SMT, the number of functional units corresponds to the number of contexts that are available. Similarly, in CMP the number of functional units corresponds to the number of processors while this number has to be halved for CMP2.

To explore the limits of performance, we briefly consider the performance of two extreme applications. First, we consider the performance of the OS delivering packets to the null application, which just drops packets upon delivery. Then, we consider the performance of the most heavily compute intensive application in our suite, namely 3DES.

Figure 3 shows the performance of each network processor executing the null application. Along the x-axis, we scale processor resources as explained above. For example, at x-axis point 4, SMT has 4 functional units shared by 4 thread contexts while CMP has 1 functional unit in each of 4 processors and CMP2 has 2 threads, 2 processors, and 4 total FUs. On the y-axis we show the number of IP packets processed per second which, in this case, indicate the peak rate at which applications can retrieve packets for processing. Since the per-packet application processing load is zero for the null application, performance is limited by how fast SPINE can hand packets off to the handler. The hand off necessitates pulling a packet off the network controller, performing some link layer processing on it and finally enqueueing it in the input queue. In our simulations using the architectural parameters of Table 4 and a high bandwidth DMA engine, this takes less than 300 cycles per packet on an SMT processor with 8 functional units totally dedicated to this task. With a 500 MHz clock rate, this translates into a peak of approximately $1.6 \cdot 10^6$ IP packets processed per second.

In our experiments, SPINE is statically assigned to one context on the SMT and one processor on the CMPs. For CMP, the peak processing rate found for SMT cannot be attained since only one functional unit is used, independently of the number of processors in the system. By assigning two functional units to the OS task, CMP2 outperforms CMP by about 25%. SMT's ability to dynamically devote resources to a given thread of execution on

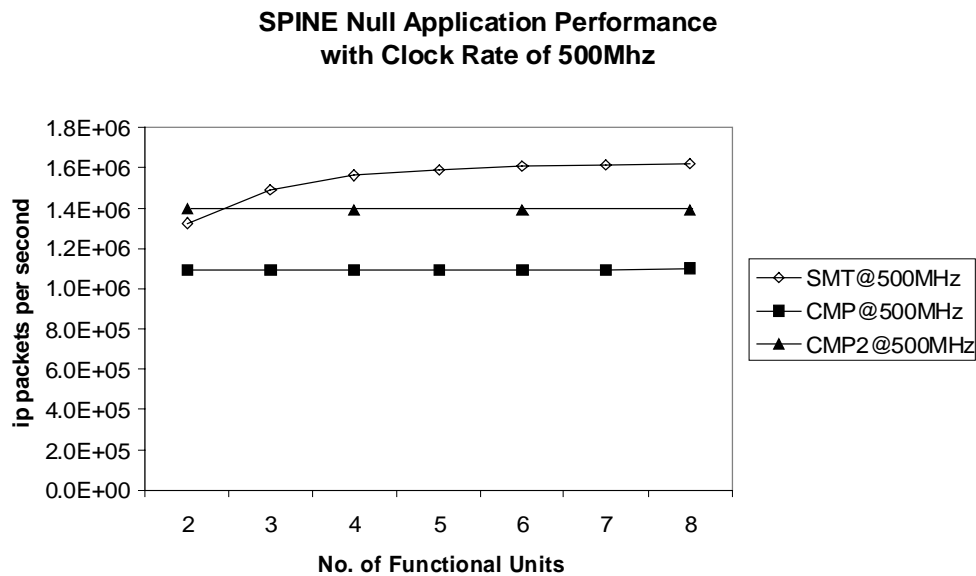


Figure 3. Null application performance. These results indicate how fast the OS can deliver packets to worker threads on each processor.

3DES with SPINE with Clock Rate of 500Mhz

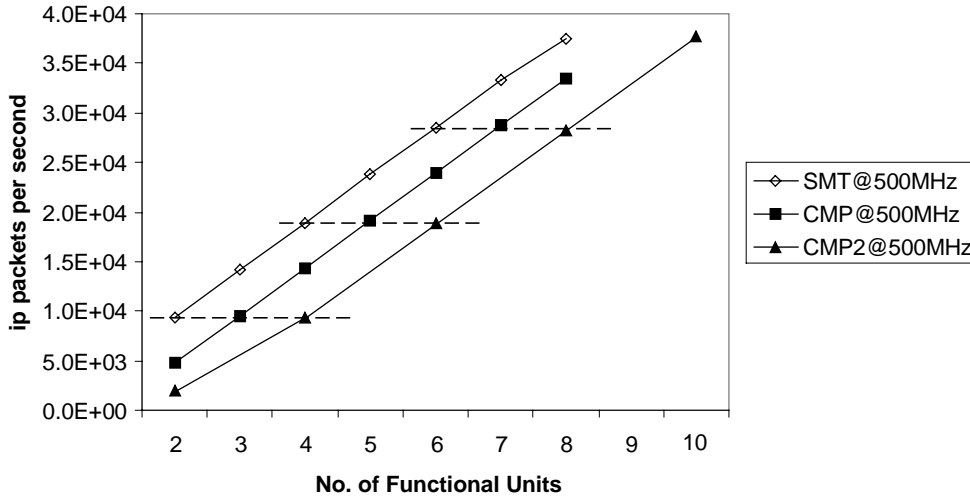


Figure 4. 3DES Performance. These results indicate the performance of each processor under a workload that executes many instructions per packet. The dashed lines show the impact of the static allocation of SPINE.

demand yields some important performance improvements up to 4 contexts. At that point, SMT is 45% more effective than CMP. Thus even if one were to take into account that CMP processors can be faster than SMT, SMT will still outperform CMP. SMT domination will carry over to applications where the per-packet processing load is light, since overall performance will be dominated by the performance of the SPINE thread.

At the other extreme, where the per-packet processing load is high, the performance of the OS thread is not so critical. Figure 4 gives the performance of 3DES on each of the architectures. As indicated in Table 1, 3DES executes a relatively large number of instructions per packet. However, the IPC for 3DES run in isolation on a sophisticated superscalar out-of-order processor (or an SMT with a single 3DES thread) is only 2.7 [7]. Figure 4 shows that using thread-level parallelism can circumvent the lack of ILP. All three architectures show a linear growth in performance. This is due to the fact that 3DES worker threads execute many more instructions than the OS thread and that they can proceed in parallel on distinct packets. Thus, it is their aggregate performance that dominates overall.

The data shown in Figure 4 leads to two other observations. First, at equal number of functional units, which in a first approximation can be construed as being a fair comparison of resources, we see that SMT outperforms CMP by about 20-25% and CMP2 by twice this amount. Thus even if we were to remove the clocking bias in favor of SMT, the latter would still perform much better than CMP2 and at least as well as CMP. Second, we can see the influence

of the OS. As shown by the dashed lines in the figure, SMT with N FUs, CMP with $N+1$ FUs, and CMP2 with $N+2$ FUs have almost the same performance. This is due to the fact that the OS pins 1 FU in CMP and 2 FUs in CMP2 without influencing in a significant manner the overall number of IP packets that can be delivered. Note that this is not necessarily a damning condemnation of the CMP and CMP2 architectures since, for engineering reasons such as practical limits in global register file size, it may be simpler to scale the number of processors in a CMP than to scale the number of thread contexts in SMT while maintaining the same clock rate.

4. Packet Filtering

The tradeoff between software and hardware-based packet classification is essentially one of flexibility versus performance. The issue of flexibility is clear: once a classifier implementation has been set in silicon on a chip, no algorithmic modifications can be made to the classifier without redesigning the part. The performance issue is less clear. How much performance overhead does a software implementation involve? In this section, we elaborate on this problem using a rather simple classification scheme since pure hardware implementations for more complex ones can be ruled out and the network processors we are investigating are not designed uniquely for extensive classification.

4.1 Classification Background

For the purposes of classification, packets are strings of bits. The packet header, at the beginning of the string, contains encoding for the packet type and the source and destination addresses, as well as other fields. Packets are composed of multiple layers of headers, each header representing a layer in the network protocol stack. Until recently, address information alone had mostly been used in routing decisions. Today, however, routing, as well as other packet processing operations, depends on different types of packet data including the packet's associated higher-level application. Corporate firewalls filtering out streaming media packets are a modern example of this. Hence, packet classifiers need to quickly look deep into a packet.

The operations involved in packet classification, also sometimes called packet filtering, are bit shift, bit masking and bit comparison. Filter specifications are written to implement the network routing and packet processing policies of an organization or network. The state machines that perform filtering can be implemented either in silicon or software. Silicon implementations are highly efficient – the circuit is designed to do this single task. A software implementation may be highly optimized, but the hardware executing the packet classification software is optimized for general-purpose tasks and is not tailored to the specific characteristics of packet streams.

Architecture	Overhead (%) at peak performance
SMT, 8 contexts	11.0
CMP, 8 cores	8.8
CMP2, 4 cores	12.4

Table 5. Software packet classification overhead using DPF.

In our system, there is a choice as to when the software based classification takes place: the OS can classify before delivering packets to worker threads, or the worker thread can classify the packet upon delivery. This distinction is unimportant for SMT since resources can be allocated dynamically as needed. For CMP, however, if the OS limits performance, as is the case when the per-packet processing load is low, it is better for the worker threads to perform the classification as the OS already has plenty to do. On the other hand, if the worker threads limit performance, as is the case for compute intensive applications, it is better for the OS to use the cycles available to it for performing classification. However, since the overhead of performing packet classification is more or less constant, the relative effect will be higher for workloads that execute fewer instructions per packet. Thus, given these issues, we choose to classify packets in the worker since for light loads it works out better for CMP (and it doesn't matter for SMT), and for heavy loads the fixed costs are negligible relative to the overall time spent processing a packet.

4.2 Results

To measure the overhead involved with software classification, we consider anew the set-up for the null application from Figure 3. When we simulate an ideal hardware implementation, we leave it unchanged, i.e., there is no overhead associated with classification. For the software case, we replace the null application by the DPF code [8] using a set of 8 filters performing 2D classification on the (protocol, port) pair in the packet header.

The results are shown in Table 5. As can be seen, the impact of the software-based classification is that the maximum rate at which packets can be handled to applications has been reduced by a factor of 9% to 12%. Alternatively, software classification can be thought of as a fixed increase in the time to process a packet. At low per-packet processing loads, the number of packets that can pass through the network processor is very high and the software classification will reduce this number. However, this reduction is less than the one encountered when running a lightweight, compute-wise, application such as IP-FORWARD [7]. For heavier computations, the effects become negligible. Thus we would rather retain the flexibility of the software solution and dedicate the silicon we save to the general-purpose processor. The remainder of the results use software-based classification.

Scenario	IP-FORWARD	HTTPMON	MD5	3DES
IP Router	56	29	7	8
Web Switch	9	63	25	3
VPN Node	3	0	70	27

Table 6. Application execution frequencies for each scenario.

5. Multi-programmed Workloads

5.1 Scenario Descriptions

In this section, we describe the three experimental scenarios used in this study. Each benchmark application (cf. **Table 2**) is used in each scenario, but in each scenario certain applications are used more than others in order to vary the load on the network processor. **Table 6** shows the average percentage of packets each application processes for the duration of the trace in each of the three scenarios. To implement these scenarios, we designed filters that classify packets from the trace according to address and higher-level (HTTP, telnet, etc.) protocols and map them to the application programs.

Although these particular scenarios, and classification policies, do not resemble in every way what one would find in a production system, the overall mix of applications and application types executing simultaneously is more realistic than looking at each of the applications in isolation as we did previously [7].

1. IP Router: mostly IP-FORWARD; light compute load

In the first scenario, the PNI is used primarily to route traffic in an internet. The device also encrypts and authenticates the subset of the traffic corresponding to the SMTP service.

2. Web Switch: mostly HTTPMON; medium compute load

The second situation simulates a PNI device used as a web switch, which monitors the state of HTTP connections for web server load balancing. The PNI in this scenario primarily functions as an HTTP monitor, and in addition continues to forward some packets and manage a virtual private network stream.

3. VPN Node: mostly MD5 & 3DES; heavy compute load

In the third and final scenario, the PNI is situated at an end-host system and is being used to decrypt and authenticate a file transfer from across the network. Most streams seen by this node belong to a VPN, and the node decrypts and authenticates those streams.

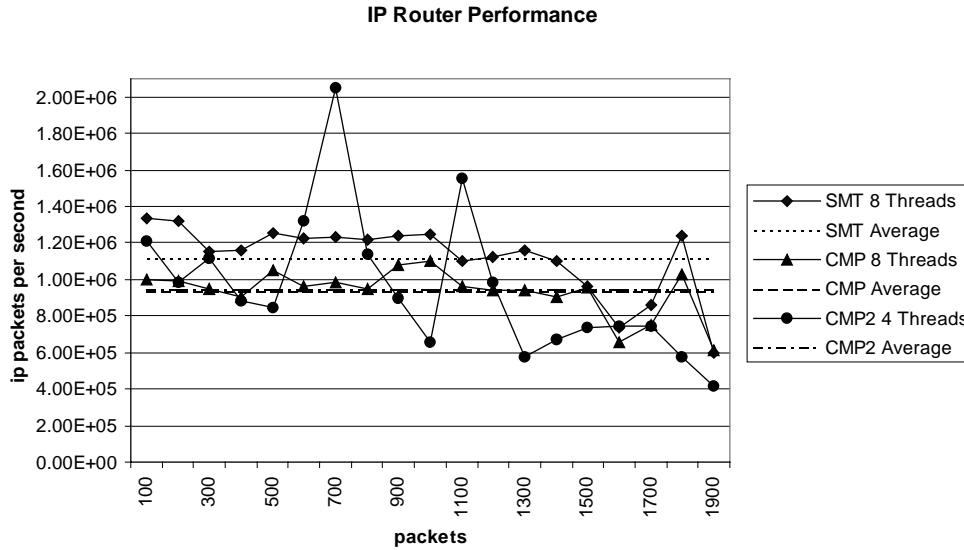


Figure 5. IP Router performance. The dashed lines indicate overall average performance, and the individual data points represent the average over 100 packets.

5.2 Results

There are two main measures of interest that we highlight in our presentation of the results. The first is meant to compare the performance of SMT, CMP, and CMP2 and we use the average number of IP packets that can be processed per second by each architecture for the comparison. In the graphs of this section, we present results for the maximum configurations (8 functional units). Recall that in some sense this favors SMT. The second metric estimates the maximum bandwidth that can be supported by a network processor architecture. In order to do so, we compute the local average of the processing time for every 100 packets over the first two thousand packets. In the graphs, the processing time is converted to the metric used in this paper, i.e., number of IP packets processed per second. In addition to indicating performance variability, the individual minimum data points for each processor (e.g., the points in the interval of packets 1900 – 2000 in Figure 5) represent the fastest network that can be supported for this workload, given the capacity to buffer 100 packets, which is a conservative assumption.

5.2.1 IP Router Performance

As indicated in Figure 5, the IP router scenario executes mostly IP-FORWARD. Since IP-FORWARD is a lightweight application, we expect SMT to outpace both CMP and CMP2, as seen with the null application performance. We see in Figure 5 that this is indeed the case although the 20% margin of superiority is less than expected for CMP. We also note that CMP2 has highly variable performance and that CMP and CMP2 achieve roughly the same average performance.

The large variation in CMP2's performance is a consequence of the fluctuating importance of the performance of the OS thread. CMP2 outperforms CMP when the OS is throttling performance, since CMP2 devotes twice as many FUs to the OS. In contrast, when the OS is not the bottleneck, CMP2 is wasting more of its FUs on the idle OS than is CMP. This is why CMP and CMP2 have opposite trends – when one performs well the other often does not. In fact, this is why the two architectures have the same average performance on this workload. Incidentally, the extreme high points that CMP2 achieves represent opportunities for better performance than that achieved by SMT. Put another way, SMT does not allocate resources in an ideal fashion; there are times when better thread scheduling decisions could be made. These peaks suggest that there is considerable benefit to be found in further work in thread/priority scheduling on SMT for this application domain.

5.2.2 Web Switch Performance

In the web switch scenario, somewhat greater computation is required of the worker threads since HTTPMON is slightly more compute intensive and MD5 is executed more often. We find in the results shown in Figure 6 that SMT does, on average, outperform the others (10% over CMP2, 20% over CMP). The over-all performance of the web switch is less than that of the IP router performance. In addition, we can see that CMP2 outperforms CMP, and that CMP2 still has variable performance, but less so than in the previous scenario.

The result of CMP2 outperforming CMP in this scenario implies that CMP does not, on average, allocate enough

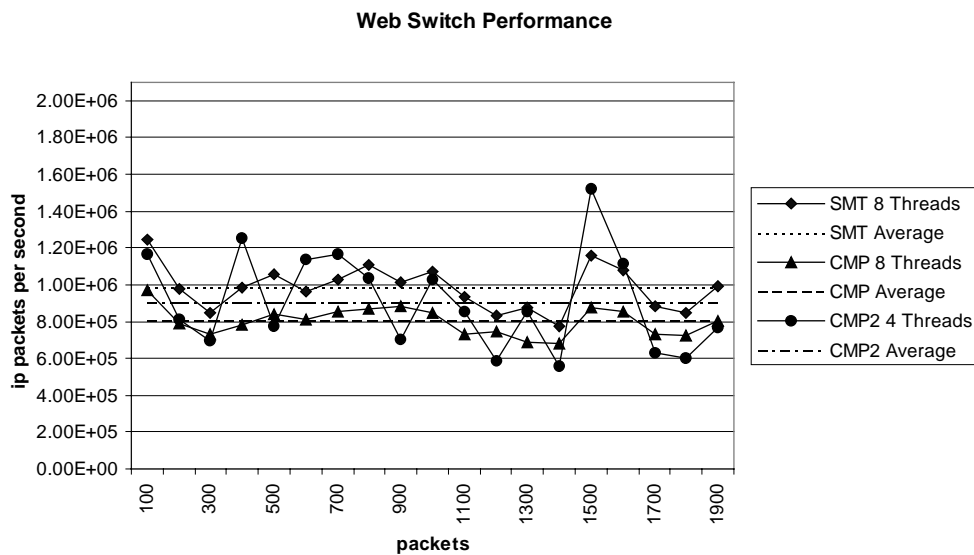


Figure 6. Web Switch performance.

resources to the OS for this workload. In allocating 2 of its 8 FUs to the OS, CMP2 better matches resources to demand. Not only is CMP2 less variable than previously, but the remaining variability is positive. That is, relative to CMP, the highs are higher than the lows are low. This suggests that CMP2's static partitioning of resources is better matched to a more compute intensive workload like the web switch scenario as opposed to the IP router scenario.

5.2.3 VPN Node Performance

Lastly, we consider the VPN node scenario. This scenario, being mostly MD5 and 3DES, is a heavyweight workload, so the aggregate performance of the worker threads is crucial. Based on the results in Figure 7, we make the following observations: (1) SMT barely outperforms CMP, (2) CMP largely outperforms CMP2, (3) CMP2 has no more variability than the others, and (4) the number of packets per second that can be processed is an order of magnitude less than in the other two scenarios.

In this final scenario, we see that CMP2 suffers from permanently devoting more of its computational resources to the OS. Performance here is dominated by the number of FUs kept busy executing MD5 and 3DES computations. CMP2, in effect, keeps two FUs idle.

We also see that CMP2 has lost all of the variability seen in earlier scenarios. This suggests that there is never a time, at least at this resolution of 100 packets, in which CMP2 outperforms CMP by devoting more resources to the

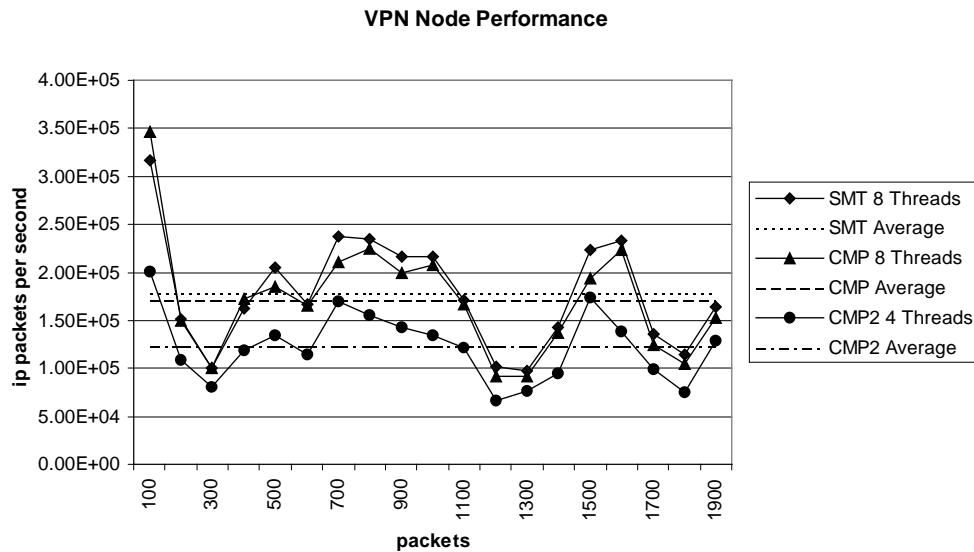


Figure 7. VPN Node performance.

OS. The partitioning is consistently inferior to CMP in this scenario.

5.3 Summary

These results have shown that SMT, as a result of its capacity for dynamically allocating resources to threads, achieves consistently better performance than CMP and CMP2 in low, medium, and high computational scenarios. Additionally, we see that there is no clear winner between CMP and CMP2. CMP outperforms CMP2 when executing heavyweight applications, and CMP2 outperforms CMP when executing lightweight applications.

A comparison between architectures on the average performance for each scenario is given in Figure 8. In addition to the previous average results, we also report the performance of CMP2 with 8 processors (denoted CMP2-8). In CMP2-8, we are doubling the total resources given to SMT and CMP2-4. While we have favored SMT when we compared it to CMP2-4, we tilt the balance the other way with CMP2-8, at least in terms of total resources. Note, however, that the resources devoted to the OS remain constant; the additional FUs are utilized by new worker threads. It is natural, in fact, to rely upon CMP's nice scalability properties to compensate for its static partitioning of resources. CMP2-8 improves over CMP2-4 by 25%, 3%, and 90% for the IP router, web switch, and VPN node scenarios, respectively. With these additional resources, CMP2-8 now achieves performance on par with, or better than, SMT. The choice between an SMT with 8 contexts and 8 functional units and CMP2 with 8 processors and 16 functional units depends on whether the additional resources in CMP2 can be traded against the more complex and more centralized design of SMT. In other words, if we want to stress hardware simplicity, we would favor scaling up CMP or CMP2. However, pinning the OS on one processor then becomes unnatural and software would become more complex, a drawback that is avoided in SMT.

In the IP router scenario, CMP2 saw large swings in performance that were a result of the fluctuating importance of the OS thread. CMP2 performed far better than CMP when the OS was the bottleneck and far worse when the worker threads were the bottleneck. When CMP2 is provisioned with 8 processors, it still outperforms CMP when the OS is the bottleneck (so the positive jumps in performance still remain). However, there are now four extra processors that come in handy when the worker threads are the bottleneck. In effect, the CMP2-8 has all of the positive variability and none of the negative variability. Consequently, average performance in this scenario is improved by about 25%.

Average Performance Comparison Between Architectures

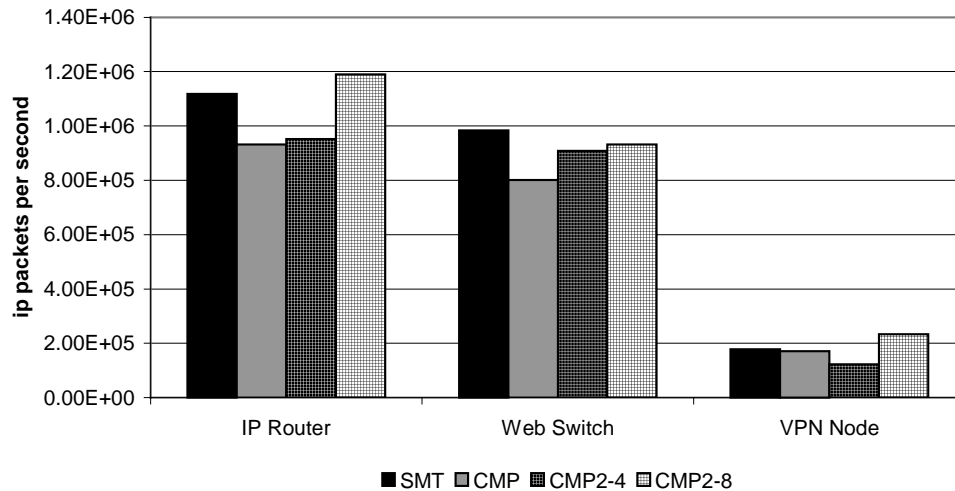


Figure 8. Average performance comparisons between architectures for each scenario, including the performance of CMP2 with 8 processors.

The improvements in the web switch and VPN node scenarios are more straightforward. The emphasis between OS and worker thread performance in the web switch was quite balanced. Hence, the additional resources devoted to worker thread processing do not result in vastly improved performance. The VPN node scenario, by contrast, was heavily constrained by worker thread performance. Therefore when worker thread resources are more than doubled, we see nearly a doubling in performance.

6. Conclusion

We have assessed the performance of multithreaded processors as the core components of programmable network interfaces. In the spirit of keeping the PNIs as general-purpose as possible, we have shown that packet classification may be implemented in software on a network processor at packet processing overheads of between 9% and 12% over an ideal hardware implementation for a simple 2D classification scheme. This overhead is a reasonable price to pay for the flexibility afforded software solutions. In the second part of this study, we have compared the performance of SMT to chip-multiprocessors with both one and two functional units per core (CMP and CMP2) for a suite of three multi-programmed network workloads with varying compute power requirements. We have found SMT to outperform the other processors, at equal resources and at equal clock rate, under all workload conditions.

SMT has great flexibility in dynamically allocating resources – and hence, for an equivalent set of resources, SMT outperforms a statically partitioned CMP over a range of workloads. The static partitioning is particularly

detrimental when the OS is pinned on a fixed set of resources. However, designing and building a large, fast CMP is easier than designing and building a large, fast SMT. If the CMP can be engineered at a greater clock frequency, or can be scaled further, then it may be worthwhile to build a faster, bigger CMP and partition resources statically. Even better would be to multithread the OS on CMP but this would come at the cost of more software complexity. The contributions of this study are intended to guide potential designers and engineers of network processors as to the nature, scale and relative importance of the tradeoffs between using SMT and CMP. As an example of a design rule, we surmise that, for heavy computational workloads in the domain of network processors, an N context SMT is roughly equivalent to a CMP with $N+1$ processors.

In addition to looking at other applications of network processors, such as load balancing and web server scheduling, further research is warranted in the scheduling of resources in CMP at the granularity of scheduling threads to processors. For example, even with static partitioning, the OS could recognize that it is idle and begin to dedicate its resources to worker threads. Similarly, we have noticed that better thread scheduling for network workloads could be achieved in SMT, perhaps using different priorities at the OS and application levels.

References

- [1] Alteon. WEBWORKING: Networking with the Web in mind. Alteon WebSystems, White Paper: www.alteon.com/products/white_papers/webworking May 3rd 1999 San Jose, California.
- [2] M.L. Bailey, B. Gopal, M.A. Pagels, L.L. Peterson, and P. Sarkar. PATHFINDER: A Pattern-Based Packet Classifier. *Proceedings of the First USENIX Conference on Operating System Design and Implementation (OSDI)*, pp. 115-224. Monterey CA, November 1994.
- [3] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzky, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. *Proceedings of the 27th Int'l Symp. on Computer Architecture*, pp. 282-293. Vancouver, B.C., June 10-14 2000.
- [4] E. Barton, J. Cownie, and M. McLaren. Message passing on the Meiko CS-2. *Parallel Computing* vol. 20, no. 4, April 1994.
- [5] A. Begel, S. McCanne, and S.L. Graham. BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture. *Proceedings of the ACM Communication Architectures, Protocols, and Applications (SIGCOMM '99)*, 1999.
- [6] N. Boden and D. Cohen. Myrinet -- A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29-36, 1995.
- [7] P. Crowley, M.E. Fiuczynski, J.-L. Baer, and B.N. Bershad. Characterizing Processor Architectures for Programmable Network Interfaces. *Proceedings of the International Conference on Supercomputing*, pp. 54-65. Santa Fe, N.M., May 8-11 2000.
- [8] D.R. Engler and M.F. Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. *Proceedings of the ACM Communication Architectures, Protocols, and Applications (SIGCOMM '96)*, 1996.
- [9] M.E. Fiuczynski, R.P. Martin, T. Owa, and B.N. Bershad. SPINE: An Operating System for Intelligent Network Adapters. *Proceedings of the Eighth ACM SIGOPS European Workshop*, pp. 7-12. Sintra, Portugal, September 1998.
- [10] A. Fox, S.D. Gribble, E.A. Brewer, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. *Proceedings of the ASPLOS-VII*, pp. 160-170, Oct. 1996.
- [11] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. *Proceedings of the ACM Communication Architectures, Protocols, and Applications (SIGCOMM '99)*, 1999.
- [12] IBM. The Network Processor: Enabling Technology for High-Performance Networking. IBM Microelectronics, 1999

- [13] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. *Proceedings of the 21st Int. Symp. on Computer Architecture*, pp. 302-313, April 1994.
- [14] LevelOne. IX Architecture Whitepaper. An Intel Company, 1999
- [15] S.S. Mukherjee and M.D. Hill. The Impact of Data Transfer and Buffering Alternatives on Network Interface Design. *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA)*, February 1998.
- [16] B.A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 67-77, May 1996.
- [17] S. Nilsson and G. Karlsson. Fast Address Lookup for Internet Routers. *Broadband Communications: The Future of Telecommunications*, 1998.
- [18] L. Peterson, S. Karlin, and K. Li. OS Support for General-Purpose Routers. *Proceedings of the HotOS Workshop*, March 1999.
- [19] J. Santos and D. Wetherall. Increasing Effective Link Bandwidth by Suppressing Replicated Data. *Proceedings of the Usenix Annual Technical Conference*, pp. 213-224. New Orleans, Louisiana, June 1998. USENIX.
- [20] Sitera Corporation. The PRISM IQ2000 Network Processor Family., <http://www.sitera.com>, 2000.
- [21] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. *Proceedings of the ACM Communication Architectures, Protocols, and Applications (SIGCOMM '99)*, 1999.
- [22] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 392-403. Santa Margherita Ligure, Italy, June 1995.