From the Proceedings of the 2000 International Conference on Supercomputing, Santa Fe, N.M., May, 2000.

Characterizing Processor Architectures for Programmable Network Interfaces

Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer, and Brian N. Bershad[*]
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195

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

Abstract

The rapid advancements of networking technology have boosted potential bandwidth to the point that the cabling is no longer the bottleneck. Rather, the bottlenecks lie at the crossing points, the nodes of the network, where data traffic is intercepted or forwarded. As a result, there has been tremendous interest in speeding those nodes, making the equipment run faster by means of specialized chips to handle data trafficking. The *Network Processor* is the blanket name thrown over such chips in their varied forms. To date, no performance data exist to aid in the decision of what processor architecture to use in next generation network processor. Our goal is to remedy this situation. In this study, we characterize both the application workloads that network processors need to support as well as emerging applications that we anticipate may be supported in the future. Then, we consider the performance of three sample benchmarks drawn from these workloads on several state-of-the-art processor architectures, including: an aggressive, out-of-order, speculative super-scalar processor, a fine-grained multithreaded processor, a single chip multiprocessor, and a simultaneous multithreaded processor (SMT). The network interface environment is simulated in detail, and our results indicate that SMT is the architecture best suited to this environment.

1. Introduction

As networks have evolved and expanded technologically, their role and importance have also increased considerably. The role of the network interface (NI), which allows a computer system to exchange messages with other systems connected to the network, has grown accordingly. To meet the functionality and performance requirements of present and emerging network applications, the current trend is to use *programmable* microprocessors on *network interfaces* (PNI) that can be customized with domain-specific software. This trend has created the so-called *network processor* market niche that many vendors, including Intel, IBM, and numerous start-ups, are eager to fill with chip architectures designed specifically to match the network application workload of PNIs. Although the architectural design of the various network processors often differs significantly, all are optimized to match the inherent parallelism present in network workloads. To date, though, there exists little performance data that allow fair comparisons between the various options. Our goal, in this paper, is to remedy this situation.

To evaluate different network processor architectures, we consider the following three questions: what workloads must the processor architecture support, what level of performance is required, and what type of architecture provides the required level of performance. In this study, we characterize both the application workloads that network processors need to support as well as emerging applications that we anticipate may be supported in the future. Then, we consider the performance of three sample benchmarks drawn from these workloads on several state-of-the-art processor architectures, including: an aggressive, out-of-order, speculative super-scalar processor (SS), a fine-grained multithreaded processor (FGMT), a single chip multiprocessor (CMP) [11], and a simultaneous multithreaded processor (SMT) [17]. After considering the benchmarks as individual standalone workloads, we then investigate the performance of each of these architectures when the applications run under an operating system designed specifically for managing programmable network interfaces [6]. In all of these performance evaluations, the key metric is the *number of messages per second* that a given architecture can support for a given workload, which translates directly into the network speeds that are enabled by the different architectures.

All of our performance evaluation is done within the context of 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 work are as follows. First, we identify a number of applications that can be used as components of a workload suite to assess the performance of PNIs. Second, we show the maximal performance attainable on these workloads (i.e., the maximum sustainable link-rate) for four high-performance processor architectures. As expected, architectures designed for a high-degree of thread-level parallelism perform best for these workloads. Lastly, we characterize the overall

[*] This work was supported in part by NSF Grant MIP-9700970, in part by DARPA Grant F30602-97-2, and by a gift from Intel Corporation.

performance of these processor architectures in the context of a programmable network interface.

The remainder of the paper is organized as follows. The next section presents background and other work related to programmable network interfaces. PNI workloads, including the benchmarks used in this study, and our experimental methodology are discussed in Section 3. Section 4 describes the high performance processor architectures evaluated in this paper. Experiment descriptions and results are given in Section 5. Finally, Section 6 concludes and proposes future work.

2. Background & Motivation

The rapid advancements of networking technology have boosted potential bandwidth to the point that the cabling is no longer the bottleneck. Rather, the bottlenecks lie at the crossing points, the nodes of the network, where data traffic is intercepted or forwarded. As a result, there has been tremendous interest in speeding those nodes, making the equipment run faster by means of specialized chips to handle data trafficking. The *Network Processor* is the blanket name thrown over such chips in their varied forms.

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 conventional application workload of such communication devices used to consist of simple packet forwarding and filtering algorithms based on the addresses found in layer-2 or layer-3 protocol packets. Today, however, the role of such devices is changing from simple packet forwarders to general-purpose computing/communications systems processing packets at layer-3 and above [13]. The application workloads used today 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].

These emerging applications require significant processing capacity per network connection. Further, 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. Unfortunately, while the 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 lookg at enhanced processor architectures.

The basic observation is that 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. It is this packet-level parallelism that can be exploited at the architectural level to achieve the sustained high-performance demanded by fast networks. Thus, it is no big surprise that network processor parts, such as Intel's IXP1200 [10], IBM's Rainier [9], and those from various startup companies, use parallel architectures, such as multiple processor cores on a single chip or fine-grain multithreaded processors, to match this packet-level parallelism. Unfortunately, there exists little data that identifies which processor architecture is best suited for the application workload used by the next generation communication devices. This lack of data and the uniqueness of the workloads motivated us to conduct this study.

3. Workloads & Methodology

This section identifies the application workloads used by network nodes, describes the evaluation methodology used to compare the performance that each processor architecture yields for these workloads, and details the execution environment that these applications are run within, which is markedly different from that of traditional operating system environments.

3.1 Workloads

There are currently three main application domains for programmable NIs, which can be roughly categorized into *Server* NI software, *Web Switching* software [1], and *Active Networking* software [15, 18]. The first two are product niches in their own right and are driving the development of programmable NIs. The latter is a research area that is pursuing the ability to dynamically deploy innovative network services rapidly and securely. With computation becoming less expensive, other NI resident applications are emerging as well.

In the context of these application domains we have identified a set of application-specific packet processing (ASPP) routines that are representative of workloads in the context of a PNI execution environment. Table 1 briefly describes present and emerging ASPPs in ascending order of per packet processing requirements.

The applications in Table 1 can be divided into two categories according to the amount of the packet that is processed. The first six applications process a limited amount of data within the protocol headers of the packet, and their processing requirements are independent of the packet's overall size. However, these applications tend to maintain state tables in complex data structures that need to be searched or accessed on a per packet basis. In contrast, the last three applications listed in Table 1 compute over all of the data contained in a packet, and therefore require a significant amount of processing capacity to process packets at the network link rate. As mentioned earlier, in these applications the processing of one packet is largely, and usually entirely, independent of the processing of any other packet. Consequently, packets may be processed in parallel.

We have selected three benchmarks from this set of workloads, namely, IP forward[*], hereafter denoted ip4, and two components of IP security, MD5 and 3DES. The first benchmark, ip4, performs address-based packet lookups in a tree data structure and is a component of conventional layer-3 switches and routers. In test runs of ip4 we perform lookups into a routing table of 1000 entries, a size representative of a large corporate (or campus) network. 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.

[*] We used algorithms for forwarding table compression and fast IP lookup described in [12].

Applications	**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[2]	Converting a multimedia data stream from one format to another within the network.
Duplicate Data Suppression[3]	Reduce superfluous duplicate data transmission over high cost links.

Table 1. Representative application specific packet processing routines. [1] **[1]**, [2] **[7]**, [3] **[14]**

Dynamic instruction characterization for these benchmarks is presented in Table 2. An examination of the instruction and data cache performance for ip4, 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%). While none of our applications tax the memory hierarchy in isolation, it is likely that cache performance and organization will be important when more than one program is running concurrently. As noted later, we will consider this fact as we evaluate the simultaneous execution of these workloads in future work.

3.2 Evaluation Methodology

As previously noted, our approach is to use the benchmarks in Table 2 to evaluate and compare the performance achieved by a number of different architectures for programmable network interfaces. Figure 1 depicts the basic components of a PNI. To drive applications during experiments, our simulator uses an IP packet trace collected from our local network to provide realistic network packet delivery to the PNI. Packets are delivered to the NI via either the host controller (out-bound packets) or the network controller (in-bound packets) and, upon message arrival, the processor is signaled. As described in the next section, the network processor then stores (into buffer memory) and processes the packet. At this point, packets may be redirected, modified, dropped or measured in some way dependent upon the target application. Since the emphasis of this study is to consider the programmability and performance of these PNI's, we focus on the network processor and its interaction with buffer memory.

To do this, we have assembled a simulation environment capable of simulating: several processor organizations (based on the SMT simulator [17], itself based on the Alpha 21x64 ISA), a full memory hierarchy including caches and main (buffer) memory, and DMA/packet transfers between main memory and a physical network link. With our simulation infrastructure, we are able to measure overall PNI system performance, in terms of packets per second, as well as many other lower level performance metrics including instructions-per-cycle (IPC), cache miss rates, prediction rates, and contention for resources such as functional units (FUs) and busses. Our simulation infrastructure provides a cycle accurate simulation of a modern high-performance, out-of-order, superscalar microprocessor core and memory system. By varying the microprocessor and memory system configurations, we can investigate a number of architectural alternatives.

3.3 Execution Environment

The execution environment for a PNI can be characterized as having a basic store-process-forward structure. 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 based on some matching criteria applied to the message. Figure 2 illustrates this structure as a pipeline of stages.

As messages arrive on the input channel, as depicted in Figure 2, they are first *stored* in the NI's buffer memory. Messages are then classified 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

Application	**Insts Executed per Message**	**Loads/Stores (%)**	**Ctrl Flow (%)**	**Other (%)**
IP forward	~200	25.4	12.7	61.9
MD5	~2000	10.7	2.8	86.5
3DES	~40000	17.8	1.2	81.0

Table 2. Benchmark characteristics.

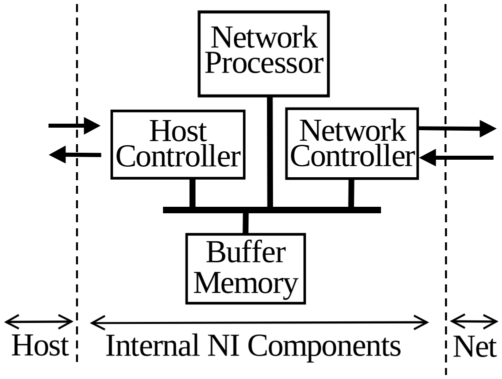

Figure 1. Generic Programmable Network Interface Architecture

schedule these concurrent operations such that each pipeline stage is occupied. Varying degrees of programmability may be used to implement each of these stages. Much work has been done to design packet classifiers in both hardware [2, 8] and software [3, 5, 16]. For this study, we assume that all packet classification is performed in dedicated hardware that is ideal in the sense that it is never the performance bottleneck in the system. The experiments discussed in this paper consider both hardware and software implementations of the packet storage stage. These implementations are generally applicable to the forward stage as well, although our benchmarks do not exercise this stage. That is, in our experiments, the PNI always acts as a consumer, or observer, of packets. From our perspective, the creation or forwarding of packets will only increase demand on the busses connecting processor, memory, and controllers. We will consider these effects in our future work.

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. [6] describes this execution environment in more detail.

4. Processor Architectures

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

Table 3. Instruction latencies.

In this section, we briefly describe the four high-performance processor architectures that we consider in this study. Unless specified otherwise, all processors may be assumed to have instruction latencies as depicted in Table 3. 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. Table 4 details the general characteristics of the four processor architectures most relevant for this study. As indicated, some of these parameters are varied in the following experiments. Our intention is to make fair comparisons between architectures possessing roughly equal resources. Therefore, to keep resources approximately equal as we scale, we provision each architecture with an equivalent amount of the primary resource, namely, the total number of functional units.

4.1 Processor Descriptions

Superscalar (SS). The SS out-of-order processors we simulate in this study have a deep pipeline (7 stages) and use scoreboarding and register renaming to resolve dynamic dependencies. The maximum number of instructions that can be issued each cycle (issue width) is equal to n, the number of functional units, which is a parameter varied in our experiments.

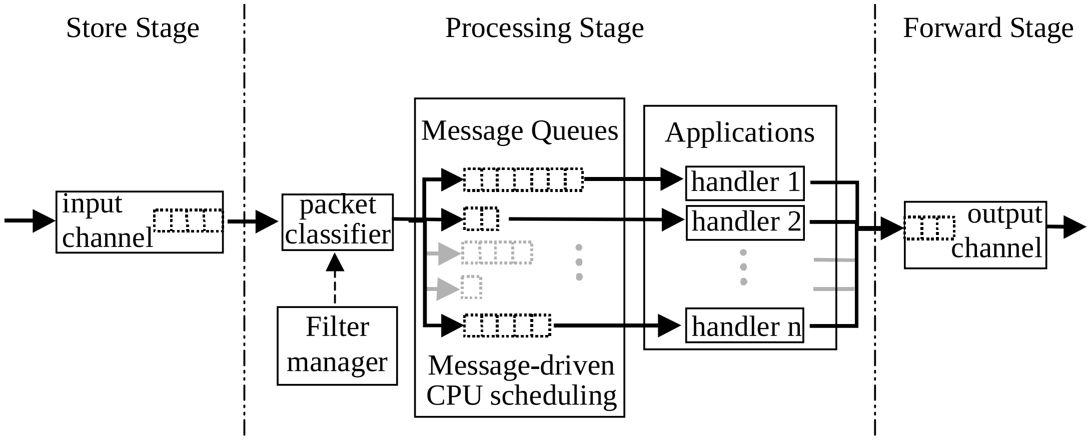

Figure 2. The Store-Process-Forward stages of messages flowing through a programmable NI.

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

Table 4. Architectural details. Cache write policies: L1 write-through, L2 write-back, no write-allocate.

Fine-Grain Multithreaded (FGMT). The multithreading support extends the core out-of-order, superscalar microprocessor by adding support for multiple hardware thread contexts. In an FGMT architecture, instructions may be fetched and issued from a different thread of execution *each cycle*. The FGMT architecture attempts to find and exploit all available ILP within a thread of execution, but can also mitigate the effects of a stalled thread by quickly switching to another thread. In this way, ideally, overall system throughput is increased. The architecture simulated here uses a round-robin thread fetch and issue policy among runnable threads (i.e., those threads in a valid state and not blocked on a synchronization event).

Chip-Multiprocessor (CMP). A CMP partitions chip resources rigidly in the form of multiple processors [11]. As an example, a 4 processor CMP has 4 separate execution pipelines, 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, each processor will have only one functional unit, and, hence, an issue width of one (therefore, the CMP will exploit no ILP). Each processor has a private L1 (both data and instruction), while a single L2 is shared amongst all processors. These caches are as described in Table 4. The L1 caches are kept coherent using an invalidation protocol similar to the one described in [11].

Simultaneous Multithreaded (SMT). An SMT architecture [17] 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 (TLP) available *between* threads. The SMT architectures we simulate in this study fetch eight instructions each cycle from a maximum of two threads.

5. Experimental Results

The experiments in this study consider the performance of each architecture while executing the benchmarks both with and without an operating system. The following subsections describe and discuss our three sets of experiments: standalone application performance, standalone operating system overhead, and OS-governed application performance.

5.1 Single Application Performance

Our first set of experiments considers the performance of each benchmark in isolation on each type of processor. Furthermore, we initially operate under the assumption that our network interface possesses "ideal" hardware for packet handling and delivery. This means that the processor will not be concerned with handling DMA requests for moving packets into and out of main (buffer) memory; this task is managed in hardware. Instead, the processor is provided with a set of buffers in memory, which are filled with packets that require application specific processing.

The processor fully commits its resources to processing those packets.

5.1.1 *Dynamic discovery of ILP: aggressive superscalar*

We first consider the performance of the aggressive superscalar processor. In this experiment, as in the others, we scale a number of architectural parameters, including clock rate and number of functional units (i.e., issue width). We scale issue width (and functional units) to show the limits of available ILP within a single thread of execution for these workloads. The overall system performance, expressed in packets processed per second, achieved by this architecture for each of our benchmarks is shown in Figure 3.

There are a number of observations to be made. First, performance for all benchmarks scales reliably with clock rate, as we expect to find with compute bound workloads. Secondly, we note that these benchmarks exhibit differing degrees of ILP, as evidenced by the fact that ip4 benefits from the addition of functional units up to the sixth, while MD5 and 3DES peak at around two and three functional units, respectively. Indeed, the aggressive SS achieved, with a maximum issue width of eight, an IPC of 2.79, 1.29, and 2.72 for ip4, MD5, and 3DES, respectively. Dramatically increasing other processor resources such as instruction queue length and renaming registers did nothing to improve performance. Hence, we may conclude that these ILP limits are properties of the workloads themselves. Given an average packet size of 128 bytes, the peak throughput of the SS architecture at 500 Mhz corresponds to network rates of 12 Gbps, 330 Mbps, and 32 Mbps for ip4, MD5 and 3DES, respectively.

These results demonstrate, and later experiments will confirm, that discovering ILP, alone, within a single thread of execution does not hold much promise towards achieving scalable, high performance.

5.1.2 *Tolerating blocked threads: FGMT*

We next consider the performance of the FGMT architecture while executing these benchmarks. An important distinction exists between this experiment and the previous one. With FGMT (and likewise for SMT and CMP), we use multithreaded versions of our benchmarks. In the SS case, the application was organized as a single thread of execution since the use of software-based threads is a detriment to performance for compute bound workloads such as ours. Our multithreaded benchmarks are symmetric in the sense that each thread executes the same code. Figure 4 graphs the performance of the FGMT architecture on each of our benchmarks. Note that in this figure, both issue width and number of thread contexts are scaled along the x-axis. We are scaling issue width just as we did in the SS case, and we are in addition adding thread contexts, i.e., $m = n$ in Table 4.

We see only a modest performance improvement over SS on MD5, while performance is nearly unchanged for ip4 and 3DES. Since there is sufficient ILP in ip4 and 3DES there is no need to hide the sorts of latencies that FGMT can help mitigate. Fine-grained thread switching was able to tolerate some of this type of delay with MD5, hence the slightly improved performance, but it did nothing to help in finding greater parallelism since at each cycle only one thread may issue. Hence, in the absence of long latency operations, performance for FGMT is limited by the ILP within a single thread and, therefore, essentially equivalent to SS performance.

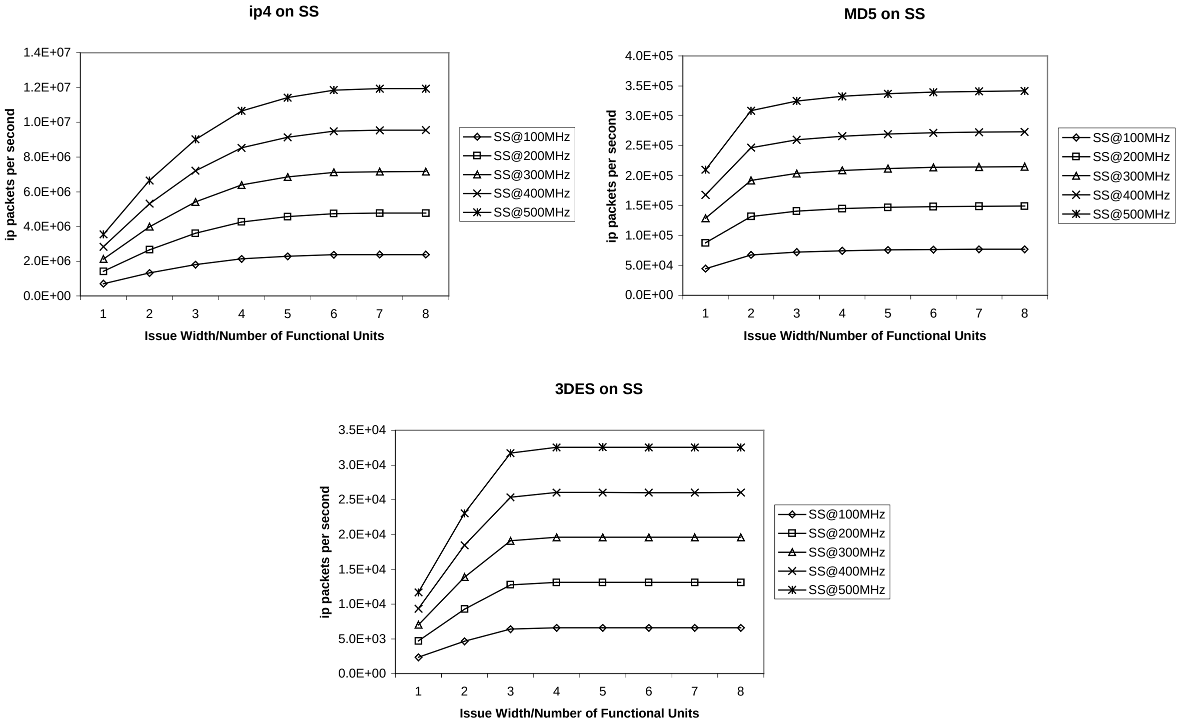

Figure 3. IP packet throughput achieved with each benchmark on a SS processor.

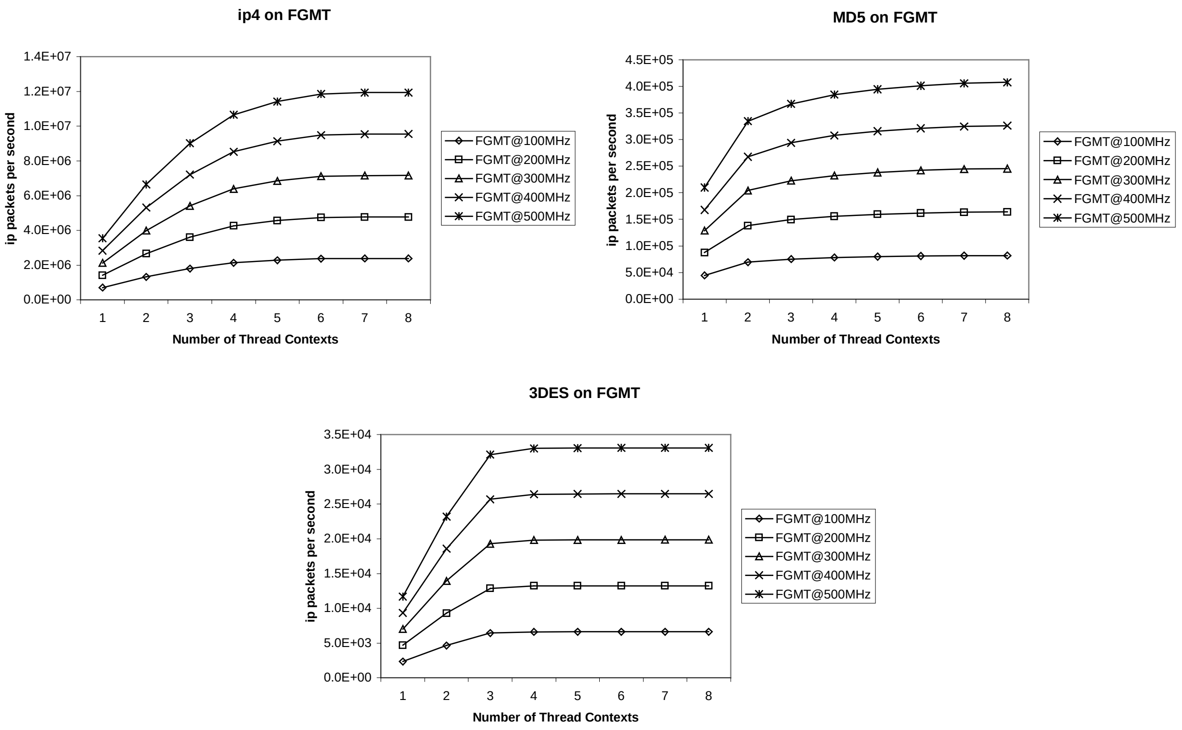

Figure 4. IP packet throughput achieved with each benchmark on a FGMT processor.

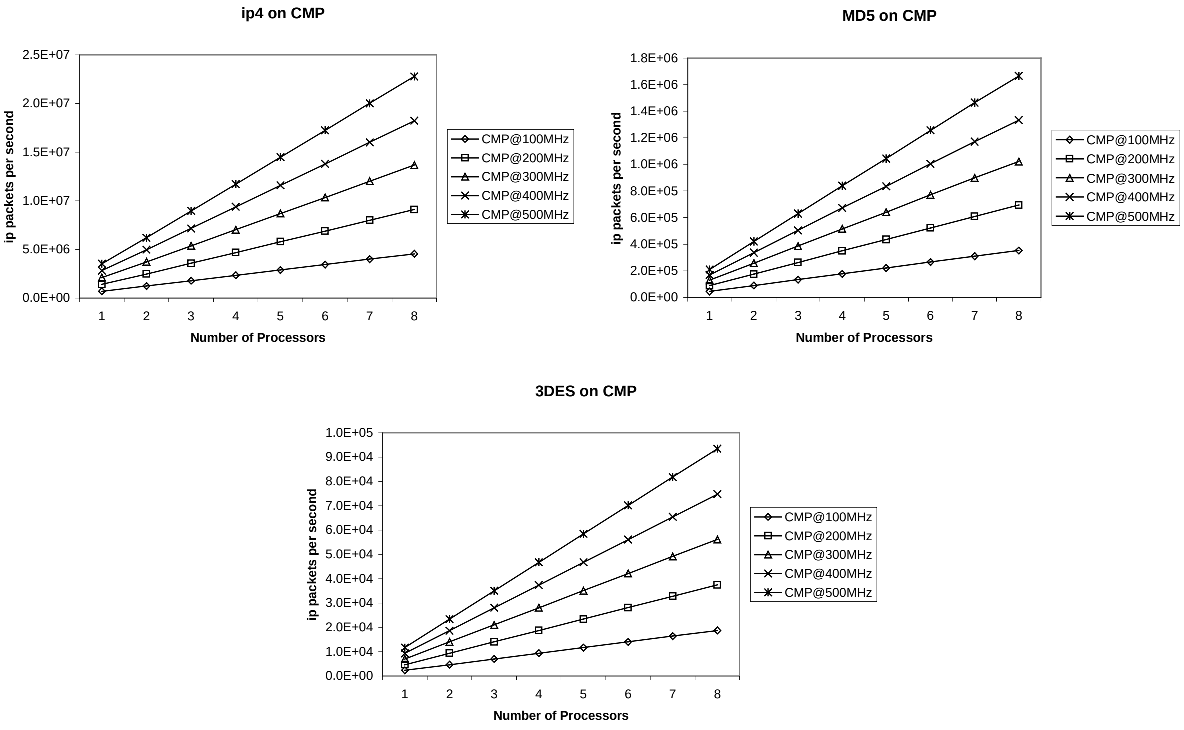

Figure 5. IP packet throughput achieved with each benchmark on a CMP.

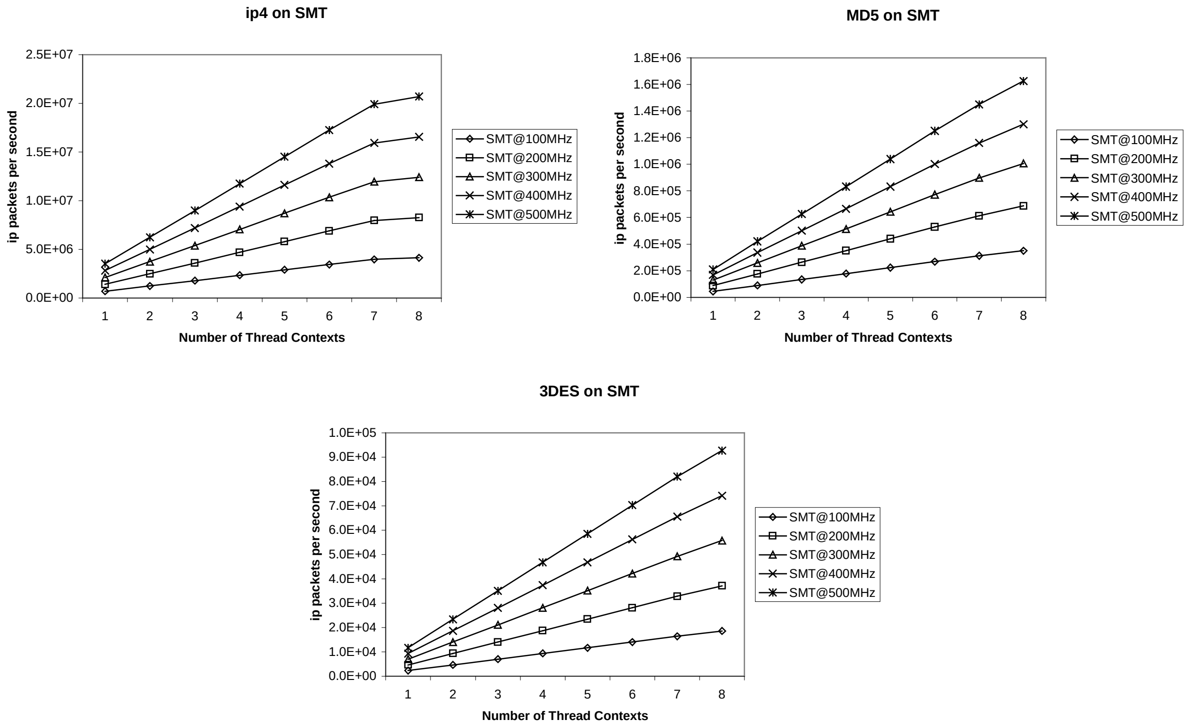

Figure 6. IP packet throughput achieved with each benchmark on an SMT processor.

5.1.3 Simple replication: CMP

An alternative approach for increasing overall IPC is to replicate the processor core on the chip, as opposed to devoting hardware resources to thread support. In our experiments regarding CMP, the processor core that we replicate has an issue width of one (a single FU) and, in the graphs, we scale the number of processor cores along the x-axis. It is important to note that we are *not* scaling the issue width of each processor core individually, but, rather, we are increasing the total chip issue width by scaling the number of single-issue processor cores. Not surprisingly, CMP performance scales linearly with the number of processors as shown in Figure 5. Despite the fact that each processor core is somewhat under-provisioned (that is, there is more ILP available in each thread of execution than the single issue core can accommodate), by replicating cores, the CMP achieves linear speedups by issuing from multiple threads simultaneously. With 8 processors, CMP achieves throughput between 2 and 4 times greater than SS.

5.1.4 ILP & Thread-level Parallelism: SMT

Finally, we consider SMT. By fetching and issuing from multiple threads each cycle, SMT combines the best features of the previous architectures, namely ILP, with thread-level parallelism. Again, in this experiment, we scale both issue width and number of hardware thread contexts. As the graphs in Figure 6 indicate, the performance of each of our benchmarks scales linearly with the number of thread contexts and issue slots. What neither the SS nor the FGMT architecture could find within a single thread of execution, SMT is able to find among all executing threads. Both CMP and SMT, that reach the same level of throughput, see linear performance increases for the same reason: they are each capable of issuing a maximum number of instructions each cycle by finding available work in all threads. It is the same characteristic that enables both, but each architecture has a different mechanism for issuing from multiple threads. Neither of the other two architectures can issue from more than one thread, and, hence, their performance is not as scalable in this manner. However, we can see that performance appears to saturate at the high-rates achieved with ip4. The SMT architecture is still being fine-tuned, and we speculate that this saturation represents contention for a shared hardware resource that has not been scaled in these experiments.

5.1.5 Comparison

These results show that SS and FGMT have basically the same performance for these workloads, and, likewise, CMP and SMT have roughly equivalent performance that is 2 to 4 times greater. For ease of comparison, Figure 7 depicts the results for all architectures clocked at 500 MHz. CMP and SMT, at clock rates of 500 Mhz, are able to sustain peak message rates of roughly 20 million, 1. 5 million, and 90 thousand IP packets per second on ip4, MD5 and 3DES, respectively. Given an average packet size of around 128 bytes, these rates correspond to network speeds on the order of 10, 1, and 0.1 Gbps. While these results clearly indicate that SMT and CMP are architectures better suited to exposing the parallelism inherent in these workloads, one could argue effectively that both the SS processor and the FGMT processor will more easily scale by clock rate, as they are less complex than SMT. This is a valid point. However, no such

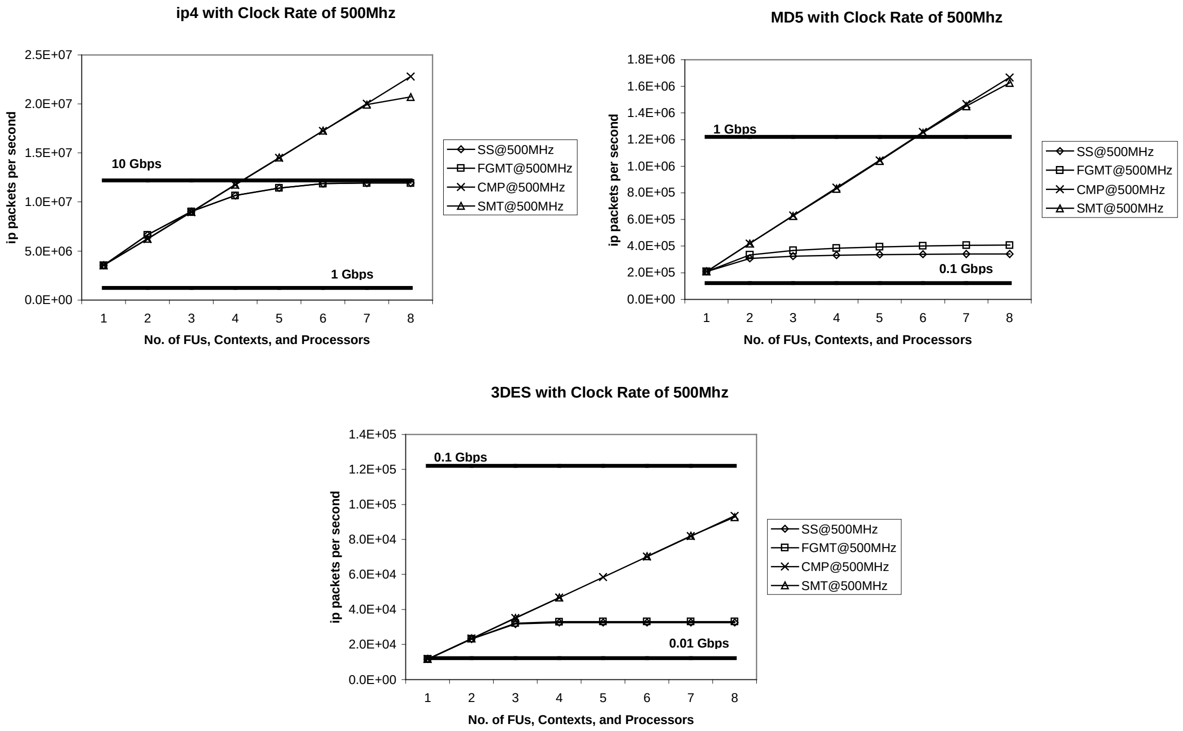

Figure 7. Performance results for all architectures, clocked at 500 MHz, running all benchmarks.

argument may be made against CMP. In the next section of results, we consider how the landscape changes as we transition from ideal packet handling hardware to OS-based packet handling in software.

5.2 Operating System Overhead

The results we have presented to this point executed in a simple test harness and assumed the presence of zero overhead operating system software and networking hardware that managed packet handling, classification, and delivery to the application processes. This section considers overall performance of these processor architectures in the context of a programmable network interface that is similar to the high-speed Myrinet [4] managed by an operating system, called SPINE, designed to enable application specific software to process messages as they stream through the system [6].

Our simulator emulates the register interface of the Myrinet packet interface and its DMA controller. As a result, the same SPINE operating system device driver code can be used either with the simulator or the actual Myrinet network interface hardware. The benefit of using the simulator is that we can study different processor architectures as well as faster network speeds. That said, we have found that the simple DMA controller used in the Myrinet incurs a fair amount of overhead when managed in software, which will become apparent in the following performance evaluation. We are considering a more sophisticated DMA controller model for use in future work to reduce the amount of software overhead incurred by the operating system.

The SPINE operating system executes in the context of a single thread. On a uniprocessor all operating system and application processing is done within that single thread context. For the multiprocessor configurations, one processor is devoted to the operating system for low latency message handling to avoid the overhead of interrupts, while the remaining processors are devoted to application processing. It is transparent to SPINE whether the multiprocessor configuration uses a FGMT, CMP, or SMT processor. A work-list based parallel computation model is used to schedule work (i.e., unprocessed packets) across the "application processors." Each of these application processors runs a single 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.

The remainder of this section describes our performance results.

5.2.1 SPINE/OS Performance

First, we determine the maximum number of packets per second that SPINE is able to deliver—the basic packet delivery rate. To do this, we execute just the operating system on the processor. That is, SPINE consumes packets from the physical network controller as quickly as possible, without dispatching any worker threads for application specific processing. The maximum packet rate SPINE is able to sustain in this situation represents the upper bound on the performance of our system since under no circumstances will worker threads be able to consume more packets than SPINE is able to produce. The results of this experiment, for processors clocked at 500Mhz, are shown in Figure 8.

As indicated, SPINE on the SS, FGMT and SMT architectures delivers packets at roughly the same rate, which is, moreover, well above the packet delivery rate sustained by SPINE on CMP. This is precisely as we would expect. SPINE runs as a single thread of execution, and, hence, has the performance characteristics of a single-threaded program. With a single thread of execution, the SS, FGMT, and SMT architectures are all more or less equivalent. Recall, however, that the CMP processor has a constant number, one, of functional units per processor core. Adding processor cores to CMP, while generally improving multi-threaded performance, does not improve single thread performance. The packet delivery rates shown in Figure 8, approximately 1.4 million packets per second for SS, FGMT, and SMT and about half of that, i.e., 0.7 million packets per second, for CMP, are therefore the performance upper-bounds that our applications are subject to while executing on SPINE.

In the best case, the above results roughly translate to 360 cycles and 720 cycles per packet on average for SS/FGMT/SMT and CMP, respectively, for packets with an average size of roughly 128 bytes. Or in other words, SPINE using a 500 Mhz processor can deliver packets to applications at a sustained rate of roughly 1.4 Gbits per second for SS/FGMT/SMT and approximately half that rate for CMP. Our previous ideal hardware results indicate that this will only be a major bottleneck for ip4, which, in a standalone fashion, processes packets above these ranges in most cases.

5.2.2 *Application Performance on SPINE*

Having determined a strict upper bound on performance, the next set of experiments will measure the performance of the three applications running on top of an operating system. This experiment differs from our previous single application experiments in that there is now operating system code that will compete with the application code for processor resources. As we have already demonstrated the effect of scaling clock rate, from this point on we report performance results only for processors clocked at 500 Mhz. The performance results for each of our benchmarks on each of the architectures is shown in Figure 9. Incidentally, in all three graphs, the first data point, which corresponds to a single functional unit, thread context, and processor for SS, FGMT & SMT, and CMP, respectively, correctly demonstrates that the architectures are equivalent for these parameter choices. Hence, each architecture's curve begins at this point. Let us consider ip4 first.

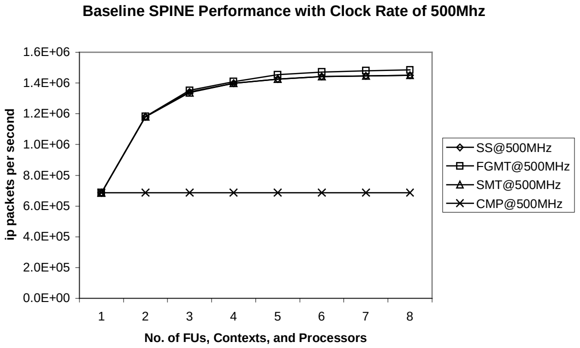

Figure 8. Basic SPINE packet delivery performance test. SS and SMT have coincident curves.

First note that for ip4, FGMT, SMT and CMP have curves similar to those seen in Figure 8. This suggests that these architectures have rather quickly run into their packet delivery bottlenecks. Further, as noted in Section 5.2.1, the ip4 application processes packets faster than SPINE delivers them. Hence, only one worker thread is ever utilized. These architectures do not achieve the peak performance shown in Figure 8 since the OS and application are now competing for resources. SS, on the other hand, performs around 1.5 times slower than its maximum. Recall that in the SS case, packet handling and application specific packet processing occur within the same thread. Hence, we see diminished results because every packet must be processed before the next one can be retrieved (these activities are overlapped in the other architectures). In many ways, ip4 is the least interesting result since the operating system represents such a severe limit to performance.

With MD5, we have a few more issues to consider. In the second column of data points, when there are two FUs, we see that SS in fact slightly outperforms SMT and CMP, which in turn slightly outperform FGMT. We have already seen that neither SPINE nor MD5 perform well with only a single functional unit. This artifact suggests that the multithreaded implementation of this application is a slightly inferior form of organization compared to the single-threaded one when the architecture is not sufficiently provisioned for *any* of the executing threads. However, we see drastic improvements for the multithreaded implementations when we increase processor resources. In contrast, SS performance settles well below that seen in the non-OS version, in Figure 3, due to the additional overhead of managing packet delivery. FGMT settles slightly below its non-OS version performance (which is considerably lower than the peak packet delivery rate) and, as before, outperforms SS by hiding instruction latencies. CMP once again settles just beneath its peak packet delivery rate. SMT, on the other hand, sees the best performance but suffers as a result of the SPINE packet producer thread competing fiercely with the worker threads. From Figure 3, we know that each MD5 thread of execution can fully utilize 2 functional units; likewise, SPINE can utilize at least 2 functional units. Here however, resources are shared more or less evenly so it may be presumed that each thread is allotted a single FU. Therefore, the performance increase diminishes greatly beyond the packet delivery rate of SPINE with one FU (approximately $7*10^5$ packets per second). It is likely that both FGMT and SMT would benefit from increasing the SPINE thread's execution priority.

With 3DES, SS outperforms the others as resources increase up to the fourth FU for the reasons discussed above regarding MD5: when there are no unused resources, serializing packet handling and processing, as SS does, is better than parallelizing these tasks. CMP and SMT achieve the linear speedups obtained in the non-OS versions shown in Figure 5 and Figure 6 as soon as there are at least two FUs. FGMT, however, exhibits terrible performance and, in fact, converges to SS performance from below. There are two reasons for this. First, performance will never exceed the non-OS result (see Figure 4), which is equivalent to SS performance and is approximately $3.2*10^4$ messages per second. Second, performance will slowly approach this value, and lag behind SS, since resources are wastefully consumed in packet handling when they would be better spent processing packets. In this case, the SPINE thread is consuming its equal share of

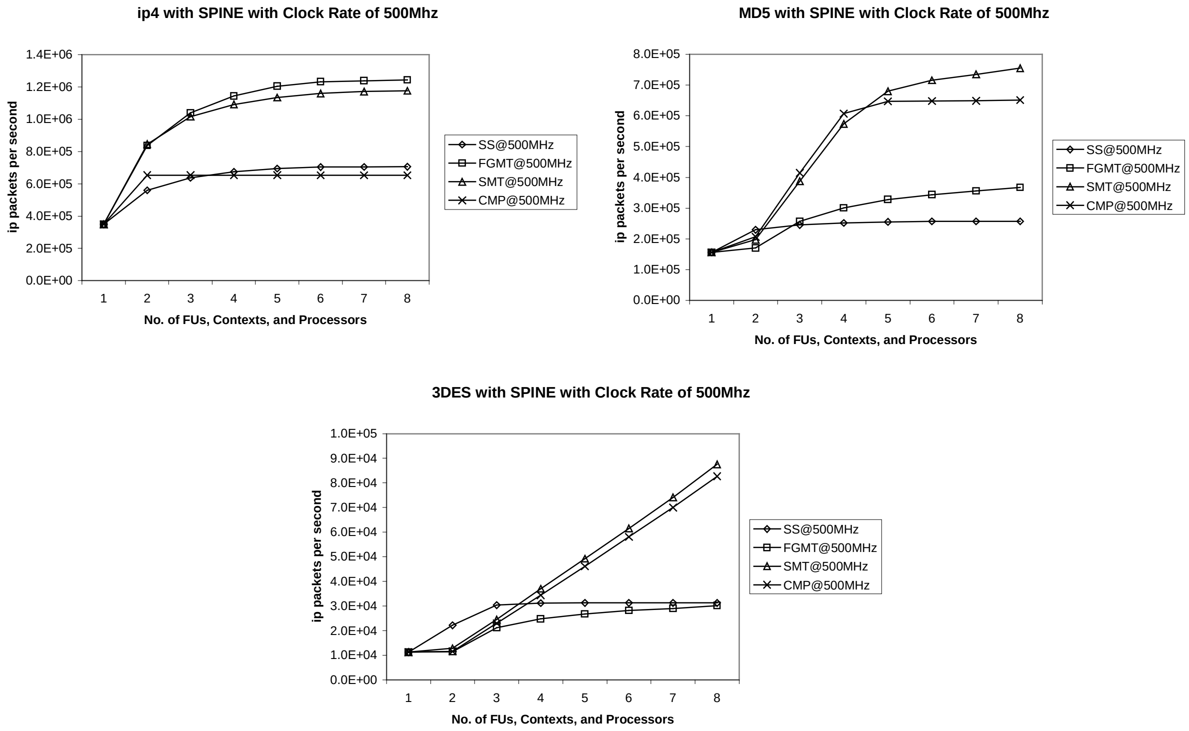

Figure 9. IP packet throughput achieved with each benchmark on each architecture, with all processors clocked at 500Mhz.

resources, as a consequence of the round-robin scheduling policy. Since 3DES is heavily compute bound, a better policy would be to give the worker threads priority over the packet delivery thread so that no resources are wasted managing packets that won't be used until the relatively distant future.

5.3 Summary

In the first set of experiments, CMP and SMT clearly demonstrate their superiority over SS and FGMT in exploiting the packet-level parallelism available within these workloads. In particular, the ability to issue from multiple threads simultaneously is key to this scalable performance. This fact is supported by the performance achieved by CMP, which exploits no ILP.

In the second set of experiments, both the operating system and applications contribute to overall system performance. Managing packet delivery and handling in software only involves a significant decrease in performance for ip4. In this case, ip4 process packets faster than the OS delivers them, hence ip4 performance on SPINE converges to SPINE performance, which is roughly an order of magnitude lower than ip4's standalone performance. MD5 and 3DES, on the other hand, process packets much slower and tend towards the standalone performance. In our simulations, SMT proves to be the architecture best suited to the workload since it is able to exploit both ILP and TLP, as no other architecture can. SPINE performance on CMP is severely limited and under-performs all other architectures since it can exploit no ILP. That said, the generally competitive performance of CMP is somewhat surprising in spite of the simplicity of each processor core.

6. Summary and Future Work

In this study, we have characterized the performance of processor architectures for programmable network interfaces. We have identified a set of benchmarks for network processors, evaluated the performance of a subset of these benchmarks on four high-performance computer architectures, and considered their performance with the added overhead from both operating system management and the underlying networking hardware. We have observed that network processor workloads exhibit a high-degree of parallelism at the packet-level, which represents an opportunity for high performance. Our experimental results suggest that, on the basis of equivalent processor resources, SMT performs better than CMP and more than a factor of two better than FGMT and SS by dynamically exploiting both instruction and thread level parallelism. For simple applications, such as IP forwarding, SMT and CMP can sustain network speeds exceeding 10Gbits/second (e.g., OC12 links). Higher speeds may be attained with better IP forwarding algorithms, such as [16]. For computationally intensive applications, such as MD5, both SMT and CMP sustain network speeds approaching 1Gbits (e.g., gigabit Ethernet).

There are a number of questions we plan to investigate in future work. First, we will augment some of the experimental architectures used in this study. The threaded architectures, FGMT and SMT, will support thread priorities to improve the thread fetch and issue scheduling algorithms. We will consider a more aggressive CMP core, with increased issue width, so that it can exploit some of the ILP available in these workloads. Secondly, we will implement additional benchmarks both from the set of more conventional applications (e.g., flow management) and emerging applications (HTTP load balancing) to round out our network processor workloads. Thirdly, we plan to implement packet classification in software to investigate the performance tradeoffs relative to hardware-based packet classification. That is, assuming that the hardware-based packet classification requires a similar number of transistors relative to a processor core, is it better to perform packet classification in software with N processors or in hardware with N-1 processors? Finally, we realize that network processors in switches/routers and host adapters will need to support some combination of these applications concurrently. Consequently, we believe that the combined behavior of these applications running in the previously described execution environment yields an interesting and compelling workload at both the system software and architectural level. Hence, we plan to measure the performance of these applications as executed in a multi-programmed environment.

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] 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.

[4] N. Boden and D. Cohen. Myrinet -- A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29-36, 1995.

[5] 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.

[6] 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.

[7] 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.

[8] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. *Proceedings of the ACM Communication Architectures, Protocols, and Applications (SIGCOMM '99)*, 1999.

[9] IBM. The Network Processor: Enabling Technology for High-Performance Networking. IBM Microelectronics, 1999

[10] LevelOne. IX Architecture Whitepaper. An Intel Company, 1999

[11] 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.

[12] S. Nilsson and G. Karlsson. Fast Address Lookup for Internet Routers. *Broadband Communications: The Future of Telecommunications*, 1998.

[13] L. Peterson, S. Karlin, and K. Li. OS Support for General-Purpose Routers. *Proceedings of the HotOS Workshop*, March 1999.

[14] 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.

[15] J.M. Smith, K.L. Calvert, S.L. Murphy, H.K. Orman, and L.L. Peterson. Activating Networks: A Progress Report. IEEE Computer Magazine vol. 32, no. 4, pp. 3-41, April 1999.

[16] 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.

[17] 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.

[18] D.J. Wetherall, U. Legedza, and J. Guttag. Introducing New Internet Services: Why and How. IEEE Network Magazine , July/August 1998.