Architectural Support for Network Applications on Simultaneous MultiThreading Processors

Kyueun Yi, Jean-Luc Gaudiot

University of California, Irvine
Department of Electrical Engineering and Computer Science
Irvine, CA 92697-2625 USA
{kyueuny, gaudiot}@uci.edu

Abstract

As network applications become increasingly sophisticated and internet traffic is getting heavier, future network processors must continue processing computation-intensive network applications at line rates. Most programmable network processors on the market today, such as the Intel IXP2800, target low performance (from 100 Mbps to 10 Gbps). However, low cost edge routers will find it hard to cope with the forthcoming sophistication of network applications to be processed at those speeds. Hence, new architectures should be designed for the programmable network processors of the future. The goal of this paper is to evaluate the applicability and efficiency of Simultaneous MultiThreaded (SMT) as a network processor. Indeed, the SMT model inherently allows the multiple parallel threads which must be dealt with in network processor applications. In this paper, we investigate the architectural implications of network applications on the SMT architecture. We demonstrate that, when executed as independent threads, applications chosen from different network layers show an improved IPC and cache behavior when compared with the situation where the program executed comes from a single network application. Finally, a new architectural solution to cope with packet dependency is proposed and evaluated.

This work is partly supported by the National Science Foundation under Grant No. CCF-0541403. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

1. Introduction

As network applications are becoming increasingly sophisticated and internet traffic is getting heavier, future network processors must continue processing computation-intensive network applications at line rates. One of the main problems is deep packet classification processing which is a major performance-critical function and is required in network applications such as QoS, URL matching, virus detection, intrusion detection, and load balancing [6]. Another is security-related processing which has become quite essential for Web switches and servers. In general, security-related processing is CPU-intensive and requires more CPU power than other network applications [9]. All network applications which require deep packet classification and security-related processing should be run at line rates, but most programmable network processors on the market today, such as the Intel IXP2800, aim at relatively low performance (from 100 Mbps to 10 Gbps). This means that such low cost edge routers will find it hard to cope with the forthcoming sophistication of network applications to be processed at line rates [7]. Hence, new architectures should be designed for the programmable network processors of the future.

Modern multiple-issue processors such as superscalar and VLIW often have more functional units than a single thread of program can effectively use. Therefore, multiple threads can be introduced so that they can share the functional units of a single processor. Simultaneous MultiThreading (SMT), a variation of multithreading, can issue multiple instructions from threads which have no dependencies among them in a single cycle and schedule them dynamically so as to concurrently exploit Thread-Level Parallelism (TLP) and Instruction-Level Parallelism (ILP) [30, 27]. Major processor manufacturers have introduced SMT in their newest architectures. These include the Compaq Alpha 21464 [5], the Intel Pentium 4 Processor with Hyperthreading Technology [14], and the IBM Power5 [24].

The data streams presented to network processors and those processed by general-purpose CPUs exhibit quite different behaviors. Indeed, the workload of network processors is inherently parallel: network packets, which are the basic unit of work for the applications, are often independent and may be processed concurrently. Network applications have also some special features such as being data-intensive, an irregularity caused by their being branch-intensive [20, 29], a high level of packet parallelism [11], and strong inter-packet dependency [19]. All these features are potentially well matched to SMT network processors which can run higher layer network applications at line rates.

Memik *et al.* [20] studied the architectural implications of network applications with nine benchmarks; their results showed the architectural implications of network applications on superscalar machines with SimpleScalar [2]. Indeed, it has been observed by Crowley *et al.* [3] that SMT could be a great candidate architecture for network applications which require high performance and parallel technology. However, the studies by Crowley *et al.* were completed with a limited number (3) of benchmarks. As discussed by Nemirovsky [1], since each network application has different architectural requirements [29, 10, 13, 20] and the throughput of network processors highly depends on the characteristics of the application, we definitely need to examine the overall architectural implications of SMT processor with as many network applications as possible if we are to design network processors with SMT as a baseline architecture.

The goal of this paper is to evaluate the applicability and efficiency of the Simultaneous MultiThreaded model as a network processor. Indeed, the model inherently allows the execution of the multiple parallel threads which must be dealt with in network processor applications. We investigate the architectural implications of network applications on SMT processors. We demonstrate that, when executed as independent threads, applications chosen from different network layers display improved IPC and cache behavior over applications chosen from the same network layer. Also, we propose and evaluate a new architectural solution to cope with packet dependency. The proposed hardware packet dependency solution has two functions. One of the functions is to choose the packets which do not occur packet dependency state in threads from packet buffer and to provide threads with the packets. The other is scheduling of the issue of Load/Store instructions to avoid packet dependency conditions in the issue stage.

The rest of this paper is organized as follows. Section 2 describes previous work on the architectural implications of network applications on single thread and multiple threads. Section 2 also includes previous work on packet dependency conditions and a quantitative analysis of packet dependency. Our simulation methodology and simulation environment are presented in section 3. We present the architectural implications of network applications on SMT processor in section 4. We then introduce and evaluate our new packet dependency solution in section 5. Finally, we summarize our observations in section 6.

2. Related work

In this section, we describe previous work on the architectural requirements of network workloads, thread synchronization and packet dependency.

The architectural demands of network workloads in single thread have been investigated by several research teams. For one, Wolf *et al.* [29] classified network workloads into packet header processing and data stream processing and presented CommBench which consists of 8 programs. The instruction frequencies, computational complexity, and cache performance of CommBench are evaluated on the SUN UntraSparc II processors operating under the SunOS5.7. Lee *et al.* [10] classified network workloads into traffic-management and quality of service group, security and media processing group, and packet processing group and presented NpBench which focused on control plane workloads. The architectural demands of NpBench is compared with those of CommBench and the computational requirements of the benchmarks with control plane functions are discussed in the paper. Memik *et al.* [20] classified network workloads into Micro-level programs, IP-level programs, and Application-level programs and presented NetBench. They investigated and compared Instruction-Level Parallelism, branch prediction accuracy, instruction distribution and cache behavior between NetBench and MediaBench with SimpleScalar. Luo *et al.* [13] presented NePSim which implements most of the functionalities of the IXP1200 and an infrastructure for analyzing and optimizing NP design and power dissipation at the architecture level. Four benchmarks were ported to NePSim and the performance was measured in terms of throughput and average packet processing time. This showed a correlation between performance variation and power variation.

Crowley *et al.* [3] showed that the performance of SMT processors was the highest compared to a Fine-Grained MultiThreaded processor (FGMT), a single chip multiprocessor (CMP), or even an aggressive, out-of-order, speculative superscalar (SS) on the basis of equivalent processor resources. However, their study used only three kinds of workloads (IP forward, MD5, and 3DES) and did not exploit the characteristic features of network applications to design their processors. Robatmili *et al.* proposed a network processor simulator based on SMT with hardware queues for scheduling process threads and some load-balancing mechanisms at the level of process threads [23]. They presented an evaluation of the performance with their own simulation environment called NPSMT, which simulates a SMT network processor, a network controller, and a packet generator. They used only two benchmark programs (IP-lookup and MD5) and used mixed scenarios of these two benchmarks.

CNP810SP is a network processor with SMT capabilities [16]. The processor can execute up to eight threads simultaneously and zero to three instructions from each of the threads at each cycle. However, it should be noted that this processor never made it into the market place. S. Melvin *et al.* proposed massively multithreaded packet processors for those "stateful" networking applications (those which are are required to support a large amount of state with little locality [18, 17]). The processor supports 256 simultaneous threads in 8 processing engines. However, no evaluation of the processor is available.

Multithreaded applications can use either coarse- or fine-grained synchronization as required. Tullsen *et al.* introduced the criteria for SMT synchronization and fine-grained synchronization using hardware-based blocking locks [28]. Two instructions, Acquire(lock) and Release(lock), were used as primitives. However, fine-grained synchronization needs significantly more programming effort than coarse-grained synchronization, even though fine-grained synchronization gives more opportunities for higher concurrency. Martínez *et al.* presented speculative locks which were based on the concepts of speculative Thread-Level Parallelism and allowed locks to be executed speculatively [15]. The speculative locks achieve fine-grained synchronization without any additional programming effort.

Melvin *et al.* have defined the concept of packet dependency condition and how it would require appropriate synchronization to correctly process packets correctly [19]. When multiple packets are being processed simultaneously in a multiprocessor or in a multithreaded environment, packet dependencies between two packets may or may not exist. When packets are processed with static rules such as in the case of stateless firewalls and forwarding engines, there are no packet dependencies because the code working in each packet does not need to modify the rules. Therefore, synchronization is not required since packets are processed independently. If packets are from the same TCP connection and it is necessary to update the state in memory for the purpose of encryption or TCP state maintenance between the processing of two packets, a packet dependency exists. Packet dependency also occurs when updating traffic management counters and routing or address translation tables. If a packet dependency exists, some sort of synchronization is required as mentioned above.

Packet dependency analysis [19] measures the probability of packet dependency of the given packet to other packets when given a certain packet window size. It shows that, with a workload considering only TCP flows, for 100 packets, there is a packet dependency probability of 14.7%. In other words, if 100 packets are currently being processed, the next packet to be received will have a dependency with 14.7 packets which are currently being processed. Packet dependency is thus becoming a critical issue if we are to achieve high performance on highly parallel network processors.

3. Simulation methodology

In this section, we describe our experimental environment which includes our SMT processor simulator and its configuration, the benchmarks and the packet traces used as input to the network application benchmarks.

3.1. SMT processor simulator with Alpha ISA

We designed an execution-driven SMT processor simulator derived from the SimpleScalar tool set [2] for our architectural investigation and our evaluation of our proposed packet dependency solution for SMT processors. We modified the *sim-outorder* simulator from the SimpleScalar toolset to implement an SMT processor simulator which supports out-of-order and speculative execution as shown in Figure 1.

The architectural processor model contains six pipeline stages: fetch, decode, issue, execute, writeback, and commit. Several resources, such as PC, integer and floating-point register files, and branch predictor, are replicated to allow multiple thread contexts. The fetch policy is ICOUNT [26] in which 8 instructions are fetched from up to 2 threads at each cycle. gShare is used as a branch prediction model. The packet scheduler and Load/Store instruction scheduler in Figure 1 are used only for the packet dependency solution and are explained in detail in Section 5. The overall configuration of the simulated SMT processor is shown in Table 1.

Although the architectural parameters in the configuration could be slightly different because of architectural variations between SimpleScalar and the Alpha processor, the configuration is designed to have an amount of resources roughly equivalent to those in Alpha EV8 [22]. The configuration for the functional units is shown in Table 2.

3.2. Benchmarks

NetBench is commonly used for the evaluation of network processors [20]. We compiled all the benchmarks with *cc* -

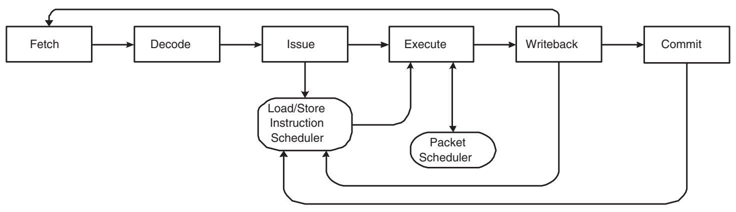


Figure 1. A Schematic diagram of our SMT processor with packet dependency solution

Parameter	Value
Fetch Bandwidth	2 threads out of 4 (8 instruction total)
Fetch Policy	ICOUNT
Decode, Issue, Commit Width	8
Branch Predictor	gShare(1K)
Branch Target Buffer	1024 4-way
Branch Mispredict Penalty	7+ cycles
Return Address Stack	32
INT Units	8
FP Units	4
Latency	L1(3 cycles), L2(13 cycles), Memory(62 cycles)
L1 I-cache	64KB(256:64:4:L)
L1 D-cache	64KB (256:64:4:L)
L2 Unified Cache	2MB (4096:128:4:L
I-TLB	1MB (1:8192:128:L)
D-TLB	1MB (1:8192:128:L)
IFQ Size	8
RUU Size	128
LQ/SQ Size	32/32

Table 1. Configuration for the simulated SMT processor

	Function	Repeat rate	Latency
INT	add/logical/shift	1	1
	mult	1	7
	div	9	12
FP	add/comp	1	4
	mult	1	4
	div	9	12
	sqrt	15	18

Table 2. Configuration of functional units

Category	Application	Arguments
Micro-Level	CRC	crc 5000
	TL	tl 32 5000
IP-Level	ROUTE	route 32 5000
	DRR	drr 32 5000
	NAT	nat 32 5000
	IPCHAINS	ipchains 10 5000
Application-Level	URL	url small_inputs 5000
	DH	dh 5 64
	MD5	md5 5000

Table 3. NetBench applications and their arguments

O3 -arch ev6 -non_shared in Alpha/Tru64 and ran each with the arguments shown in Table 3.

We implemented a minor modification of NetBench-1.1.0 to compile it in Alpha/Tru64 since some network-related header files of Alpha/Tru64 are slightly different than those of gcc/Linux. To remedy the unaligned access memory error which is caused by improper casting at run-time, we implemented a minor modification to ROUTE, DRR, NAT, and URL. We also modified four benchmarks, TL, ROUTE, DRR, and NAT, to use all four bytes of IP address in the routing table instead of the first byte of the IP address. The original NetBench uses the traces from Columbia University available in the public domain [21]. However, destination and source IP addresses of this trace are anonymized for privacy protection. We used other real packet traces [4]. The DH benchmark, Diffie-Hellman public-key encryption-decryption mechanism, does not need a packet trace.

4. Implications of network applications on the design of SMT processors

In this section, we investigate the implications and performance of NetBench on SMT processors. To demonstrate how well SMT architectures fit network processors, we compare the performance between the execution of one thread and the execution of the same three threads chosen from each benchmark of NetBench, and examine the performance of the execution of three threads chosen from the different levels shown in Table 3.

4.1. Architectural requirements of workloads consisting of the same applications

In this subsection, we examine what kind of performance enhancement can be observed with the SMT processor. We compare three performance parameters, IPC, the Level-1(L1) data cache miss rate, and the Level-2 (L2) unified cache miss rate between the execution of one-thread and the execution of the same three-threads from NetBench. According to an earlier study [20, 10], even a 4KB L1 instruction cache shows a comparatively low 2.8% miss rate and our experiments show virtually 0% miss rates with a 64KB L1 instruction cache in both one-thread experiments and three-threads experiments. Thus, we do not consider the L1 instruction cache miss rate for performance comparison since the miss rate of the L1 instruction cache does not depend on any specific feature of network applications in NetBench but mostly depends on a function of the cache size. However, we need to investigate whether the L1 instruction cache miss rate of the *next* generation network applications such as deep packet processing and security related processing depends on any specific feature of those next generation network applications.

Figure 2 shows the IPC comparison between the execution of one-thread and the execution of the same three-threads chosen from each benchmark program of NetBench. The average IPCs of the execution of one-thread and the execution of three-threads are 1.65 and 2.64, respectively. As intuitively expected, the SMT processor brings a 60% performance enhancement in network applications.

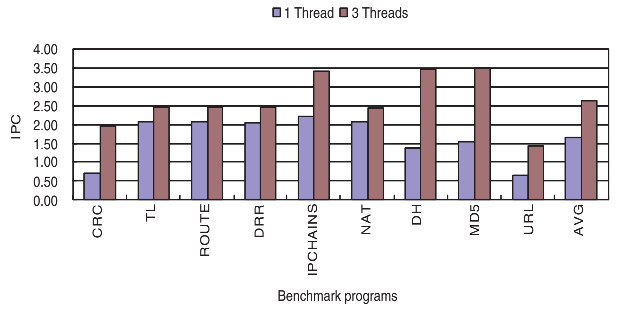


Figure 2. IPC comparison between 1 thread and 3 threads

Figure 3 shows a comparison of the L1 data cache miss rates between the execution of one-thread and the execution of three-threads. The averages of L1 data cache miss rates for the same cases are 1.8% and 3.2%, respectively. As the number of threads is increased, the L1 data cache miss rates are increased by 77%. The reason for this increase is inherited from the SMT processor in which threads share the L1 data cache and inter-thread conflict misses occur. The L1 data cache miss rates are increased as the number of threads increases, since the SMT processor shares the cache hierarchy among all the threads [12, 8]. More particularly, in the network workloads such as TL, ROUTE, DRR, and NAT, which are data-intensive and based on the routing table, the L1 data cache miss rates are increased by 125%. This means that data-intensive network applications have less locality in the L1 data cache than any other network applications which do not use routing tables in NetBench. URL, a particularly branch-intensive network application, shows much higher L1 data cache miss rates than any other network application.

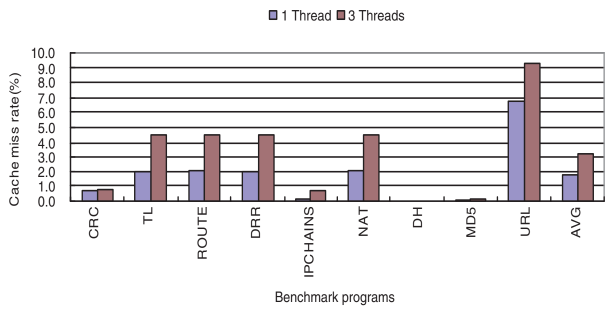


Figure 3. Comparison of L1 data cache miss rates between 1 thread and 3 threads

Figure 4 shows a comparison of L2 unified cache miss rates between the execution of one-thread and the execution of three-threads. Usually, the L2 unified cache size is sufficiently large to store the working set of multiple threads, hence, L2 unified cache miss rates are slightly higher [12, 8] as shown by the data-intensive network workloads such as TL, ROUTE, DRR, and NAT of the Figure 4. However, the averages of L2 unified cache miss rates of execution of the same cases are 25.0% and 9.4%, respectively. The averages L2 unified cache miss rates are higher than the averages L1 data cache miss rates. The L2 unified cache miss rates of computation-intensive network applications such as CRC, IPCHAINS, DH, and MD5 are significantly higher than the L2 unified cache miss rates of any other network applications in the execution of one-thread. This means that computation-intensive network applications have less locality than any other network applications in the L2 unified cache. This is the reason for the increase of the average of L2 unified cache miss rates. When comparing the average L2 unified cache miss rates between the execution of one thread and the execution of three threads, the average L2 unified cache miss rates executing three threads is smaller.

4.2 Architectural demands of mixed-style workloads

Network applications are spread across several layers and network processors need to execute them concurrently in

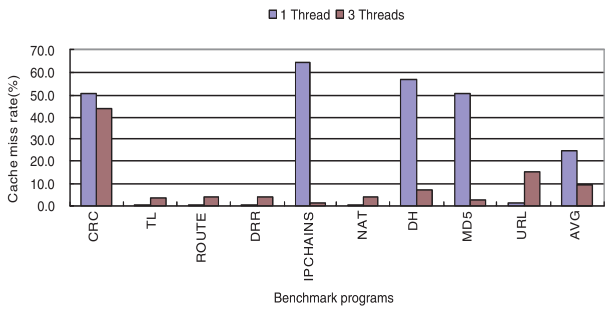


Figure 4. Comparison of L2 unified cache miss rates between 1 thread and 3 threads

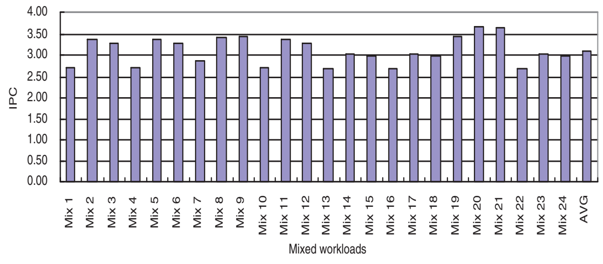


Figure 5. IPC of mixed-style workloads

order to keep the processing at line rates. Network applications of NetBench as shown in Table 3 are classified into 3 categories: the micro-level, the ip-level, and the application-level; we choose one application from each level, form a total of 24 mixed-style workloads, and run them on the SMT processor simulator. Hence, we observe the execution of mixed-style workloads as shown in Table 4.

Name	Benchmarks	Name	Benchmarks
Mix1	crc, route, url	Mix13	tl, route, url
Mix2	crc, route, dh	Mix14	tl, route, dh
Mix3	crc, route, md5	Mix15	tl, route, md5
Mix4	crc, drr, url	Mix16	tl, drr, url
Mix5	crc, drr, dh	Mix17	tl, drr, dh
Mix6	crc, drr, md5	Mix18	tl, drr, md5
Mix7	crc, ipchains, url	Mix19	tl, ipchains, url
Mix8	crc, ipchains, dh	Mix20	tl, ipchains, dh
Mix9	crc, ipchains, md5	Mix21	tl, ipchains, md5
Mix10	crc, nat, url	Mix22	tl, nat, url
Mix11	crc, nat, dh	Mix23	tl, nat, dh
Mix12	crc, nat, md5	Mix24	tl, nat, md5

Table 4. Mixed-style workloads

Figure 5 shows the IPC of these mixed-style workloads. The average IPC is 3.18 which is an increase of 21% over the execution of the same three threads and of 93% over the execution of one thread. It is caused by the fact that mixed-style workloads reduce resource conflicts in the pipeline since different network applications have different instruction mixes [20, 10] and there is a strong likelihood that different resources would have been required.

Figure 6 shows the cache miss rates of the mixed-style workloads. Compared to the execution of the same three threads, the average L1 data cache miss rate is not changed but the average L2 unified cache miss rate is reduced by approximately 26%. Hence, mixed-style workloads can increase the IPC by 21% and reduce the L2 unified cache miss rate by 26% compared to the execution of the same workloads. However, it should be noted that the average L1 data cache miss rate is not changed with mixed-style workloads.

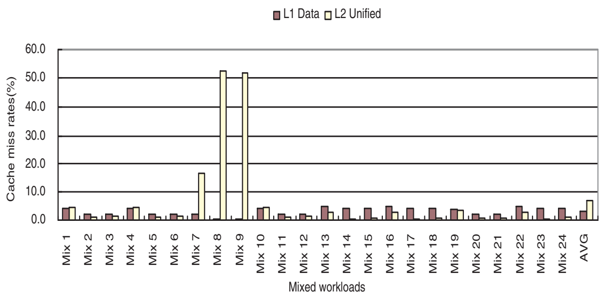


Figure 6. Cache miss rates of mixed-style workloads

4.3 Architectural demands of network applications on the SMT Processor

As mentioned above, network workloads are spread across several layers and network processors need to execute them concurrently. Thus, mixed-style workloads are a good fit for SMT processors. Indeed, mixed-style workloads showed better performance in terms of IPC and L2 unified cache miss rate. However, mixed-style workloads do not have any influence on the L1 data cache miss rate of data-intensive network applications. Thus, we must somehow reduce the L1 data cache miss rates of data-intensive network applications and the L2 unified cache miss rates of computation-intensive network applications.

5. Packet dependency solution

Several synchronization schemes for multi-threaded/multiprocessor architectures have been introduced in section 2. Packet dependency which occurs during packet processing in the network applications and which requires some form of synchronization has also been introduced in section 2. In this section, we present previous packet dependency solutions and propose and evaluate a novel hardware-based packet dependency solution. It consists of packet schedulers and of a Load/Store instruction scheduler.

5.1 Previous packet dependency solutions for multiprocessors and multithreading

The problem of packet dependency can be approached in hardware or in software [19]. In the software approach, software locks are inserted in the code to hold the processing of dependent packets until the dependent packet is fully processed. However, inserting locking mechanisms in the packet processing code may be difficult since the code is usually large and the correct insertion of locking mechanisms is hard to verify.

In the hardware approach, Figure 7 (a) shows the packet classification hardware which enforces sequentiality to all packets and groups the flow for multiple processing elements. In such environments with multiple processing elements, the hash function sends all packets of the same flow to the same processing engine. However, the packet classification hardware can degrade the performance since it sequentially processes all packets, but even network applications do not require sequentiality. The packet classification hardware also suffers from the need to balance work across the multiple processing elements. As a solution to the problem of packet classification hardware, Melvin *et al.* [19] proposed a conceptual hardware solution which consists of a Read table, a Write table, and an Active packet list. The hardware solution is located between the packet processing engine and the memory system. When the hardware finds the packet dependency status, it sends a stall signal to the packet processing engine which should be stalled until the dependent packet which is currently under processing by another packet processing engine is completed.

5.2. Our proposed packet dependency solution for SMT processors

As described in section 2, a packet dependency occurs (*i*) when packets being processed are from the same TCP connection and (*ii*) when updating the shared traffic management counters and routing or address translation tables. As a solution to these problems of previous packet classification hardware, we propose a novel packet dependency solution which consists of a hardware packet scheduler and a hardware Load/Store instruction scheduler as shown in Figure 7 (b). The hardware packet scheduler prevents the occurrence of condition (*i*) and the hardware Load/Store instruction scheduler prevents the occurrence of condition (*ii*).

Packet scheduler: The hardware packet scheduler is located between a thread (considered a processing element) and a packet buffer. Each thread in an SMT processor has its own hardware packet scheduler with its own packet buffer. The number of packet schedulers is the same as the maximum number of threads in an SMT processor. When each packet reaches the network processor, packets are distributed to each packet buffer evenly. For example, if there are three threads, packets are stored in the packet buffer as shown in (b) of Figure 7. Rectangles with the same color in the packet buffer of (a) and (b) in Figure 7 mean that they belong to the same TCP connection. We consider only the TCP protocol in this paper since the protocol of the traces we used for this experiment is TCP. When each thread asks for its own packet scheduler to read a packet, the packet scheduler takes one candidate packet from its own packet buffer. Then, the packet scheduler determines whether the candidate packet has the same TCP connections as the packet being processed by other threads. If the candidate packet has the same source/destination address and source/destination port as the packet being processed by other threads, the packet scheduler makes a decision that the candidate packet has a packet dependency. Thus, the candidate packet is skipped and the packet scheduler examines the next packet in its own packet buffer. If the next packet does not have a packet dependency with other packets being currently processed by other threads, the next packet is sent to the thread. The packet so skipped is examined again when the thread reads a packet in the next time. The proposed packet dependency solution does not suffer from the problem of having to balance work across the multiple threads since packets are distributed to each packet buffer evenly. Since the proposed packet dependency solution does not require packet sequentiality, the proposed packet dependency solution does not suffer from performance degradation which is caused by packet sequentiality.

Load/Store instruction scheduler: In our baseline SMT processor, Load/Store instructions are issued to functional units with register renaming property such as Register Update Units (RUU) [25] when their operands are ready. The Load/Store instructions which access the shared memory such as routing or address translation tables should be scheduled carefully: this should be done so as to access the shared memory sequentially in order to prevent differ-

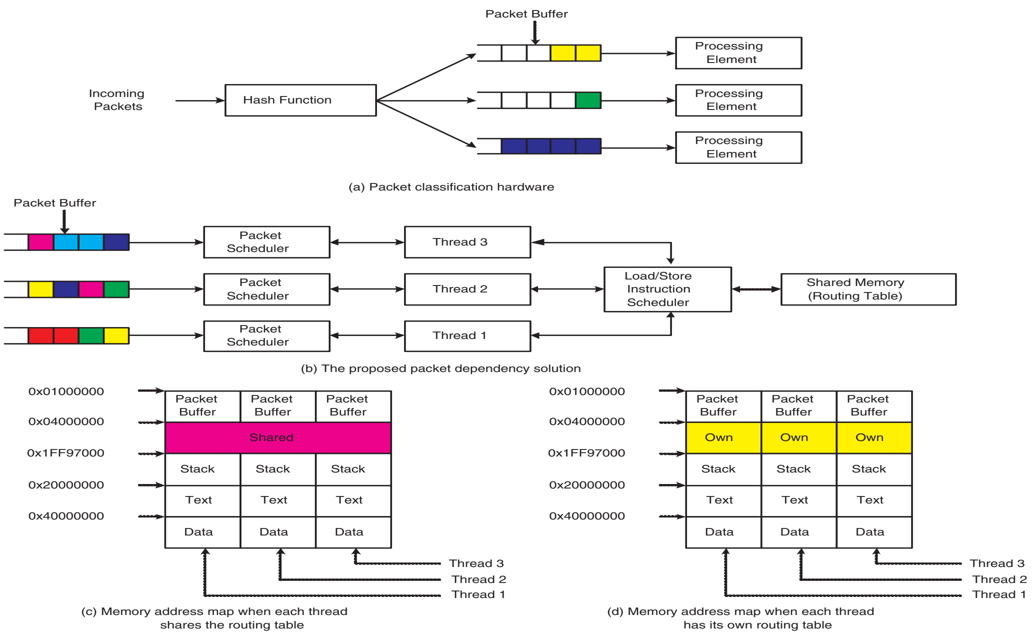


Figure 7. The packet classification hardware, proposed packet dependency solution, and memory address map

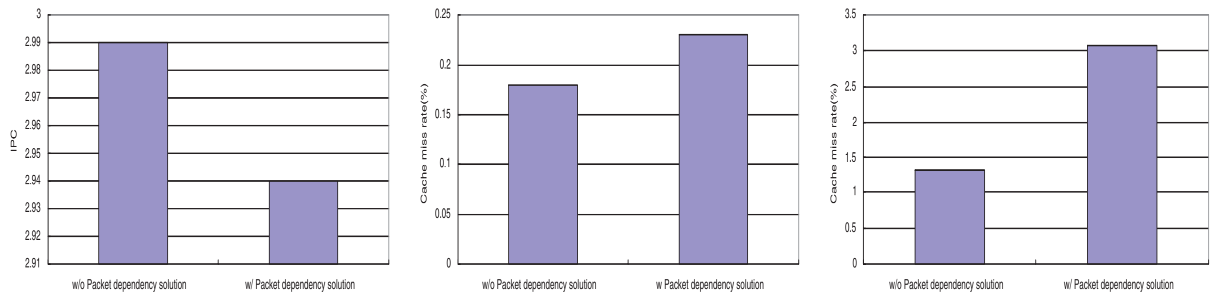
ent threads from accessing the same location at the same time. The proposed Load/Store instruction scheduler is located between a thread and a shared memory. The proposed Load/Store instruction scheduler has a circular queue and works with 3 stages out of the 6 pipeline stages in our SMT processor: issue, writeback, and commit. In the issue stage, whenever Load/Store instructions are issued, the Load/Store instruction scheduler examines the address to which the Load/Store instruction refers. If the Load/Store instruction attempts to access the shared memory, it first examines the circular queue. If the circular queue is empty, this Load/Store instruction is placed in the front location of the circular queue and continues to access the shared memory. If another one is in the front location, it means that the other instruction accesses the shared memory, the instruction is placed in the rear location of circular queue and waits in issue stage. The instruction which is located in the right after front location of the circular queue will locate in the front location of the circular queue as the previous instruction in the front location of circular queue is removed from the circular queue. Thus, the instructions can access the shared memory when they are placed in the front location of the circular queue. In the commit stage, when the Load/Store instruction is completed to access the shared memory, the Load/Store instruction in the front location of circular queue is removed, which enables Load/Store instruction in the right after front location of the circular queue to locate in the front location and access the shared memory. In the writeback stage, Load/Store instructions which must be removed in the issue stage because of branch misprediction are removed from the circular queue. The proposed SMT processor has the memory address map as shown in (c) of Figure 7.

5.3. Evaluation of the proposed packet dependency solution

To evaluate our proposed packet dependency solution, we compare the performance of two experiments: in the first one, each thread shares one routing table with the proposed packet dependency solution. In the other, each thread has its own routing table. With this comparison, we investigate how much the hardware of the proposed packet dependency solution influences the performance.

For the experiments, the TL benchmark of NetBench is used and each experiment runs three threads. The experiment with the packet dependency solution starts with the routing table which is stored in the shared memory as shown in (c) of Figure 7. The experiment without packet dependency solution starts with the routing table which is stored in its own memory as shown in Figure 7 (d).

As shown in Figure 8 (a), the IPCs of the experiments without the packet dependency solution and those with the packet dependency solution are 2.99 and 2.94, respectively.



(a) IPC (b) L1 data cache miss rate (c) L2 unified cache miss rate

Figure 8. Performance comparison between without packet dependency solution and with packet dependency solution

The hardware which works as a packet dependency solution and is added to a normal SMT processor degrades the performance by only 1.7%. The Figure 8 (b) shows the L1 data cache miss rates. The L1 data cache miss rates of experiments without packet dependency solution and with packet dependency solution are 0.18% and 0.23%, respectively. The hardware of the packet dependency solution increases the L1 data cache miss rate by 28%. Figure 8 (c) shows the L2 unified cache miss rate. The L2 unified cache miss rates of experiments without packet dependency solution and with packet dependency solution are 1.32% and 3.06%, respectively. The hardware of the packet dependency solution increases the L2 unified cache miss rate by 132%.

Since the Load/Store instruction scheduler influences the behavior of the caches, as we expected, the performances of the L1 data cache and L2 unified cache are degraded. However, the performance of our proposed packet dependency is competitive since the IPC is only slightly decreased. Since our SMT processor simulator is derived from the SimpleScalar toolset which does not support a synchronization function, we cannot easily compare the performance between the proposed packet dependency solution and the previous software packet dependency solution at this moment.

6. Conclusions

While work on architectural implications of network applications on single thread [29, 10, 20, 13] and on architectural implications of network applications on multiple threads [3, 23] can be found, past work on architectural implications of network applications on multiple threads used only a few applications as benchmarks from many network applications. As mentioned in the introduction, each network application shows significantly different architectural implications. We definitely need to examine the overall architectural implications with as many network applications as possible in order to design network processors with SMT as the baseline architecture.

In this paper, we have investigated the architectural implications with all 9 benchmarks of NetBench in SMT processors. More specifically, we have found that mixed-style workloads in which network workloads were chosen from different network levels showed better performance than the same workloads chosen from the same application. The mixed-style workloads increase the IPC by 21% and reduce the L2 unified cache miss rates by 26% compared to the same workloads.

We have proposed and evaluated a packet dependency solution for SMT processors. The proposed strategy consists of packet schedulers and of a Load/Store instruction scheduler. Since it does not require packet sequentiality and grouping of the flows, it does not exhibit any performance degradation which would be caused by packet sequentiality. Besides, since packets are distributed to each packet buffer evenly, the proposed packet dependency solution does not suffer from the need to balance work across the multiple threads. The hardware which works as a packet dependency solution and is added to a normal SMT processor decreases the IPC by only 1.7%. The performance of the proposed packet dependency is competitive since the IPC is only slightly decreased. The proposed packet dependency solution can be used in multiprocessors or CMPs with only slight modifications.

As future research, we will compare the performance between our proposed packet dependency solution and traditional synchronization used in multithreaded environments. We will develop benchmarks which represent next generation network applications such as deep packet processing and security-related processing and evaluate them in order to investigate the architectural implications in SMT processors.

References

[1] A. Nemirovsky. *Towards Characterizing Network Processors: Needs and Challenges.* XSTREAM LOGIC Inc., White Paper, November 2000.

[2] D. Burger and T. M. Austin. *The SimpleScalar Tool Set, Version 2.0.* http://www.simplescalar.com.

[3] P. Crowley, M. E. Fiuczynski, J.-L. Baer, and B. N. Bershad. Characterization Processor Architectures for Programmable Network Interfaces. In *Proceedings of the 2000 International Conference on Supercomputing*, 2000.

[4] D. E. Comer. *Computer Networks and Internets with Internet Applications, 4th edition.* Prentice Hall, 2004.

[5] K. Diefendroff. Compaq chooses SMT for Alpha. *Microprocessor Report*, 13(16):1–7, 1999.

[6] F. Gebali and A. N. M. E. Rafiq. Processor Array Architectures for Deep Packet Classification. *IEEE Transactions on Parallel and Distributed Systems*, 17(3):241–251, March 2006.

[7] Intel. *Intel IXP2800 Network Processor*.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach, 3rd edition.* Morgan Kaufmann, 2003.

[9] K. Kant, R. Iyer, and P. Mohapatra. Architectural Impacet of Secure Socket Layer on Internet Servers. In *The Proceedings of the 2000 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 7–14, Austin, Texas, USA, September 2000.

[10] B. K. Lee and L. K. John. NpBench: A Benchmark Suite for Control plane and Data plane Applications for Network Processors. In *Proceesings of 21st International Conference on Computer Design(ICCD'03)*, pages 226–233, 2003.

[11] H. Liu. A Trace Driven Study of Packet Level Parallelism. In *Proceedings of International Conference on Communications*, 2002.

[12] J. L. Lo, S. J. Eggers, , J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems*, August 1997.

[13] Y. Luo, J. Yang, L. N. Bhuyan, and L. Zhao. NePSim: A Network Processor Simulator with Power Evaluation Framework. *IEEE Micro Special Issue on Network Processors for Future High-End Systems and Applications*, pages 34–44, Sept/Oct 2004.

[14] D. T. Marr, F. Binns, D. L. Hill, G. Hilton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.

[15] J. F. Martínez and J. Torrellas. Speculative Locks for Concurrent Execution of Critical Sections in Shared-Memory Multiprocessors. In *Workshop on Memory Performance Issues (WMPI), at International Symposium on Computer Architecture (ISCA)*, Gothenburg, Sweden, Jun 2001.

[16] S. Melvin. *Clearwater Networks CNP810SP Simultaneous Multithreading (SMT) core.* http://www.zytek.com/ melvin/clearwater.html, 2000.

[17] S. Melvin. *Flowstorm prothos massive multithreading (MMT) packet processor.* http://www.zytek.com/ melvin/flowstorm.html, 2003.

[18] S. Melvin, M. Nemirovsky, E. Musoll, J. Huynh, R. Milito, H. Urdaneta, and K. Saraf. A Massively Multithreaded Packet Processor. In *NP2: Workshop on Network Processors, held in conjunction with The 9th International Symposium on High-Performance Computer Architecture*, Anaheim, California, February 2003.

[19] S. Melvin and Y. Patt. Handling of Packet Depenencies: A Critical Issue for Highly Parallel Network Processors. In *Proceedings of International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2002.

[20] G. Memik, B. Mangione-Smith, and W. Hu. NetBench: A Benchmarking Suite for Network Processors. CARES Technical Report No. 2001-2-01, 2001.

[21] Passive Measurement and Analysis project, National Laboratory for Applied Network Research. http://moat.nlanr.net/Traces.

[22] R. P. Preston *et al.* Design of an 8-wide Superscalar RISC Microprocessor with Simultaneous Multithreading. In *International Solid-State Circuits Conference (ISSCC)*, page 334, San Francisco, CA, February 2002.

[23] B. Robatmili, N. Yazdani, and M. Nourani. Optimized SMT processors for IP-packet processing. *Microprocessors and Microsystems*, 29:337–349, 2005.

[24] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[25] G. S. Sohi. Instruction Issue Logic for High-Performance, Interruptable, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[26] D. M. Tullsen, S. J. Eggers, J. S. Emer, and H. M. Levy. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simulataneous Multithreading Processor. In *Proceedings of 23rd Annual International Symposium on Computer Architecture*, 1996.

[27] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[28] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999.

[29] T. Wolf and M. Franklin. COMMBENCH - A Telecommunications Benchmark for Network Processors. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, pages 154–162, Austin, TX, Aprial 2000.

[30] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirovsky. Performance Estimation of Multithreading, Superscalar Processors. In *Proceesings of the 27th Annual Hawaii Internation Conference on System Sciecces*, volume 1, pages 195–204, 1994.