PERFORMANCE ANALYSIS OF CACHING EFFECT ON REAL TIME PACKET PROCESSING IN A MULTI-THREADED PROCESSOR

by

MIAO JU

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINERRING

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2007

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervising professor, Dr. Hao Che, for his patience, encouragement and advice during my research. His strong understanding of the subject and constant motivation, made this work possible. I would like to thank him for giving me the opportunity to work in his research group. I would also like to thank Dr. Mohan Kumar and Dr. Jeff Lei for being on my committee.

I take this opportunity to thank all my friends at UTA. Especially thank Hun Jung, Anil Kumar and Rugu in our lab for their valuable discussions and support. I also would like to thank Mr. Xing Zhao from Columbia University and Mr. Yang Liu from SUNY at Buffalo for their helpful suggestions.

Finally, I would like to thank my parents for their constant encouragement and support without which this work would have not been possible.

July 10, 2007

ABSTRACT

PERFORMANCE ANALYSIS OF CACHING EFFECT ON REAL TIME PACKET PROCESSING IN A MULTI-THREADED PROCESSOR

Publication No. ______

Miao Ju, M.S.

The University of Texas at Arlington, 2007

Supervising Professor: Hao Che

Caching has been time proven to be a very effective technique to improve memory access speed and average performance for general processors. Based on the real-world trace simulation, earlier research showed that cache can help improve the route lookup and packet classification performance in a Network Processor (NP). However, the existing studies did not take the packet delay/loss constraints into account. As a result, how effective the caching technique is, in dealing with traffic under stringent delay/loss constraints (as is the case for router interface using an NP for packet processing), is still an open issue.

In this thesis, we aim at addressing the above issue through simulation studies based on a well-designed, lightweight simulator. We first demonstrate how such a simulator can be developed to allow effective performance analysis of a multi-threaded, single core processor. Then we apply this simulator to the study of the caching effect on the packet throughput performance under various delay/loss constraints. Our simulation studies indicate that the effectiveness of caching is sensitive to the actual delay/loss constraints. When drop/loss constraint is loose, use of a larger number of threads can effectively hide the memory latency, making caching less effective. Moreover, the effectiveness of caching is getting worst, when drop/loss constraint is tight and the number of threads is relatively small. Finally, our simulation shows that when cache miss distribution is uniform, caching is effective, improving throughput performance by 4.6% even when the miss ratio is as large as 27.2%.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

With the enormous momentum behind Internet-related technologies and applications, demands for data network bandwidth are on the rise at an astounding rate. As a result, the amount of throughput required by network processors (NPs) is increasing significantly. For example, supporting OC-192 (10 Gbps) and OC-768 (40 Gbps) line rates require that a packet be processed within 52 nanoseconds and 13 nanoseconds, respectively.

There are two critical functions in packet processing, i.e. IP forwarding table lookup (or IP-lookup) and packet classification. Most packet processing algorithms are based on some utility data structures. These data structures involved in routing tasks are stored on-chip in large SRAMs with latencies in the range of 10 to 30 cycles. Thus, much of the time spent on packet processing is taken up by reading (and sometimes writing) of the data structures in the SRAM multiple times, creating bottlenecks and non-deterministic performance for packet processing. A widely recommended solution is to employ cache to speed up IP-lookup and/or packet classification.

In fact, most existing studies claim that caching is effective in improving both IP-lookup and packet classification performance and that the caching technique should be adopted in NP for real-time packet processing [15, 16, 17, 18, 19, 20, 21]. However, in this thesis, we argue that the reasoning behind such a claim is inadequate for the following two reasons. First, such a claim is mainly based on the ground that the cache miss ratios were low for real-world traffic traces tested (0.1%-27.2% for IP-lookup and 3%-25% for packet classification). How the cache misses will impact the packet delay and loss performance has not been tested (note that packet delay and loss constraints are generally tight for real-time packet processing). Second, to make the testing environment as close to the real-world environment as possible, most existing results were obtained based on either real NP or cycle-accurate simulation (with emulated caching) and real-world traffic traces as input. There are two problems with this approach. First, simulation based on real NP or cycle-accurate simulation is generally time consuming and slow. Second, the real-world traffic traces used in the simulation generally run at low line rates and may not represent future traffic demands. As a result, the claim was drawn based on very limited samples in a large parameter space, which is clearly inadequate.

For the above reasons, we conclude that it is still an open issue as to whether or not caching is effective in improving real-time packet processing performance. The need to quantify the benefits and drawbacks of caching in support of real-time packet processing becomes even more urgent, as multi-core, multithreaded processors have increasingly been adopted in router interface cards for packet processing, which heavily rely on caching for performance enhancement.

1.2 Scope and Contributions

This thesis aims at quantitatively characterizing the benefits and drawbacks of caching for real-time packet processing in a multithreaded, single-core NP/processor. Unlike the existing work, this thesis characterizes caching effects in a large parameter space and based on multiple performance measures. This allows the impact of caching effect on the overall packet processing performance to be comprehensively characterized. This thesis makes the following two major contributions.

First, we build a lightweight processor simulator, which can be easily adapted to any multithreaded, single-core processor architecture. We then demonstrate that this simulator can provide packet throughput/latency data within 10% of the cycle-accurate simulation results. Moreover, this simulator is extremely fast and requires only a pseudo code as input for simulation (note that cycle-accurate simulators require micro-code as input for simulation). As a result, it allows effective performance analysis of a multi-threaded, single-core NP/processor in a large parameter space.

Second, we apply this simulator to the study of the effect of caching for IP-lookup on the packet throughput performance. The simulator emulates a typical IP packet forwarding using an IXP1200 NP with the addition of caching for IP-look. The simulation study is performed in a large parameter space in terms of hit ratio, number of threads, and line rate, and under various delay/loss constraints. As a result, it successfully characterizes the benefits and drawbacks of caching as a potential mechanism to improve real-time packet processing performance.

1.3 Thesis Organization

The remaining part of this thesis is organized as follows. Chapter 2 gives an overview of the related work. Chapter 3 describes a lightweight simulator for multithreaded, single-core processors. Chapter 4 discusses the simulation results for caching based packet processing. Finally, chapter 5 provides conclusions and future work.

CHAPTER 2

RELATED WORK

2.1 Processor Simulators

The traditional processor/NP simulation tools are aimed at faithfully emulating the processor microscopic processes, and are useful for fine tuning the processor configuration to achieve optimal performance. They are not designed to allow fast processor performance testing.

The most popular processor simulator is an execution driven simulator called "SimpleScalar". SimpleScalar [5] is a tool which can simulate the behavior of a general purpose processor based on SimpleScalar architecture. The architectural characteristics studied in [5] would be more applicable to MIPS-based processor architecture rather than RISC-based architecture (used by most of the modern day processors). Moreover, SimpleScalar needs to know the exactly instructions of the code, and is very slow.

There has been a few efforts to study the behavior of multithreads in network processors. Most existing NP simulation tools are aimed at providing rich features to allow detailed statistical or per packet analysis. One such effort was explained in [2] where the performance of two types of architectures – single processor with multithreading (SMT) [2] and chip-multiprocessors (CMP) [2] was analyzed. The simulation used a cycle accurate simulator [3] [4] with multi-programmed work load [2]. The work load [4] comprised of three different tasks – IP forwarding, a web-switch monitoring HTTP requests and connections and VPN node that performs encryption/decryption and authentication. The main contribution of [2] was the comparison of the performance of SMT and CMP processors. Cycle accurate simulator is very slow and memory hungry. Even for the most lightweight NP simulator described in [14], it is reported that it takes a hour to simulate 1 second of hardware execution on a Pentium III 733 PC with 128 Mbytes memory, assuming the micro-code is available as input to the simulator.

System level modeling tools like POOSL [6] and Click-modular router [7] [8] have also been used for modeling NPs. [9] compared the results from an analytical model to that of the simulation on an Intel IXP 1200 network processor and the results are shown to be within 15% accuracy.

In [7] the authors presented several examples of modeling uniprocessor and multiprocessor systems executing IPv4 routing and IPSec VPN encryption/decryption applications. The performance results of the architecture in Click-modular [7] model were compared to the actual results measured on the real systems being modeled; the results were found to be accurate within 10%.

[10] and [11] estimated the performance of an NP for different applications. It provided a new scheme to estimate end-to-end packet delays, packet queuing and to explore the design spaces. This scheme can be used to quickly develop new architectures which can be later analyzed in detail using other design tools. [12] and [13] described a systematic approach to benchmarking NPs.

2.2 Caching

Recently, a lot of research work has been done in performance analysis of caching effect on packet processing. Some studies [22] [23] discussed if there is sufficient temporal locality to justify the use of cache. There are two major functions in packet processing, IP-lookup and packet classification. [15, 16, 17, 20, 21] show the caching effect in route lookup, [18] and [19] demonstrate how a caching technique can help improve the performance of packet classification.

However, almost all of these researches are based on the real world trace simulation. B. Talbot et al. [22] studied the packet traces captured from enterprise class routers. Keith Morris et al. [15] uses three different packet traces collected from the sole router that connects Taiwan's academic network to the ISP in California. Tzi-cher Chiueh et al. [17] gets packet traces from public domain. Kang Li et al. [18] uses traces obtained from an OC-3 link that connects a university campus network to the Internet. We note that the line rates in these traces are low. For example, OC-3 link trace in [16] and [18], 9Mbps Internet link in [19], and 45Mbps Internet link in [17].

2.2.1 Caching for IP Lookup

The IP forwarding table which is usually stored in an on-chip SRAM consists of a set of entries, each containing a destination network address, a network mask and an output port identifier. NPs need to go to the SRAM several times to get the final lookup result. Earlier research has shown that caching frequently accessed entries in an IP forwarding table can help improve the IP-lookup performance in a NP based on the real-world trace simulation. Recently research works concentrate on three areas:

1. Trying to find out the cache design tradeoffs, by tuning time, space and configuration parameters [15] [20] [21].
2. Developing new efficient data structure or hashing function to speed up the lookup process [16]
3. Using appropriate cache update mechanism to reduce the miss ratio [17].

2.2.2 Packet Classification Processing

Li et al. [18] demonstrated how a cache's associativity, replacement policy, and hash function contribute in varying magnitudes to the cache's performance. Specifically, they concluded that small levels of associativity can result in enormous performance gains; that replacement policies can give modest performance improvements for under-provisioned caches; and that the faster, less complex hashing can improve overall cache performance.

Chang et al [19] examined the accuracy issue in cache design space. In particular, they quantify the benefits of relaxing the accuracy of the cache on the cost and performance of packet classification caches.

CHAPTER 3

SIMULATOR DESIGN

3.1 Processor Organization

This research will consider a generic processor organization depicted in Fig. 1. In this organization, a processor is viewed generically as composed of a set of core components, i.e., core and a set of on-chip or off-chip supporting components, such as I/O interfaces, memory, special processing units, scratch pads, embedded CPUs, and coprocessors. These supporting components may appear at three different levels, i.e., the thread, core, and processor levels, collectively denoted as *mem*, *Mem*, and *MEM*, respectively. Each core supports multiple threads which are scheduled based on a given thread scheduling discipline. Cores may be configured in parallel and/or multi-stage pipeline (a two-stage configuration is shown in Fig. 1). Data path functions that are mapped to a given processor will be further mapped to different cores at different pipeline stages or different cores at a given stage. Clearly, this generic processor organization covers most existing processor architectures.

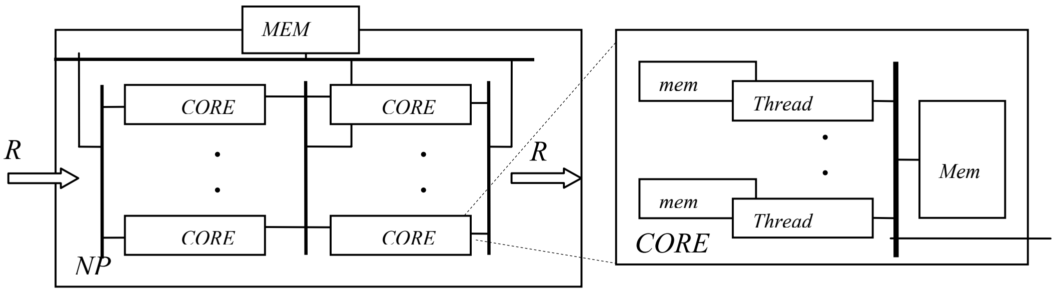

Figure. 1 A generic processor organization

3.2 Code Path

An important concept used in processor performance analysis is code path. We explain how we define the code path in this research work through an example. Fig. 2 gives a simplified flow diagram or graphical representation of the pseudo code for the data path functions to be processed in a typical router data path, including IP forwarding, MPLS label swapping, and the IS-IS routing protocol processing. An incoming packet is in the form of an Ethernet frame. The core first inspects the EtherType field in the Ethernet header to identify the upper layer data format for the frame payload. There are four possible outcomes:

(1) It is an IS-IS routing protocol packet. In this case, the frame is sent to the control card without further processing;

(2) It is an IP packet. In this case, the IP forwarding is performed which may include firewall/policy filtering, DiffServ traffic conditioning, IP forwarding table lookup, TTL (i.e., Time-to-Live) update, checksum update, and so on. Then the layer 2 framing is performed on the packet which may include outgoing interface MTU check, packet fragmentation, ARP table lookup, and layer 2 framing.

(3) It is an MPLS encapsulated IP packet. In this case, the MPLS label swapping table lookup is performed. As a result, there are two possible outcomes, i.e., the packet needs to be label forwarded or IP forwarded downstream. In the former case, the label is swapped and the layer 2 framing is performed on the labeled packet. In the latter case, the label is popped off and the IP forwarding in case

(2) is performed.

(4) It is an unknown protocol. In this case, the frame is simply discarded.

Now, assume that all the data path functions are mapped to a single core. Then a unique branch from the root to a given leaf is defined as a *code path* associated with that core. An incoming packet to a core is always associated with one code path. Fig. 2 didn't expose the branches that may exist in each processing module. For example, in the MPLS label swapping module, there are a number of branches corresponding to different numbers of label popping and pushing. So the number of code paths is much larger than what we have seen in Fig. 2.

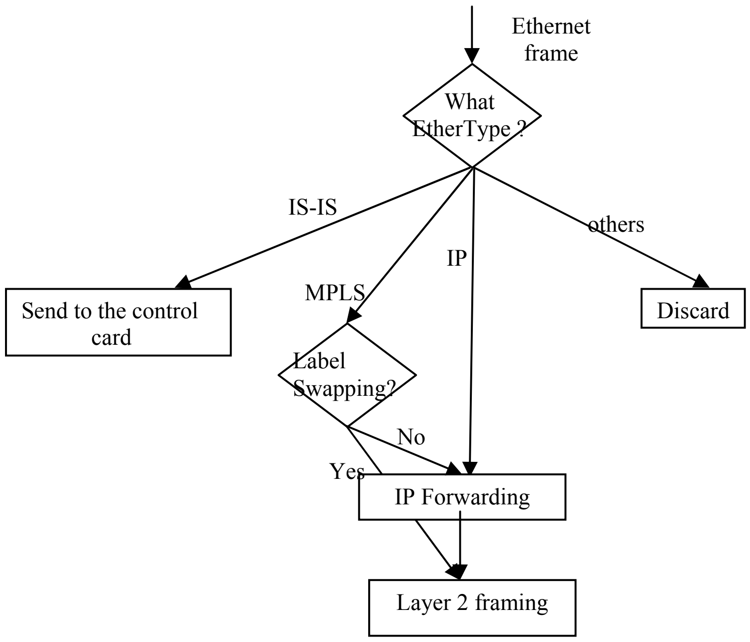

Figure. 2 A pseudo code or flow diagram for fast data path functions typically seen in a router interface

Now, we formally define a code path as a sequence of instructions that a core has to execute throughout the life-time a packet is handled by a thread in that core. A code path may be broken down into smaller sequences of instructions due to supporting component accesses. For example, for a code path that involves IP Forwarding, an IP forwarding table lookup in an off-chip TCAM coprocessor may be performed in the middle of the code path. Hence, a code path can be generally expressed in the following format:

$T_k(M_k;\ m_{1,k},\ t_{1,k},\tau_{1,k},\ \ldots,\ m_{Mk,\ k,},\ t_{Mk,\ k},\tau_{Mk,\ k})$: Code path k with access to $m_{i,k} \in mem,\ Mem,$ *or MEM,* and unloaded access latency $\tau_{i,k}$ after the $t_{i,k}$-th cycle

In the code path, where $k = 1,\ \ldots,\ K$ and $i = 1,\ 2,\ \ldots,\ M_k$, where M_k is the total number of supporting component accesses.

$|T_k|$: code path length or total number of instructions in the code path $T_k(M_k;\ m_{1,k},\ t_{1,k},\ \tau_{1,k},\ \ldots,\ m_{Mk,\ k,},\ t_{Mk,\ k},\tau_{Mk,\ k})$, where $k = 1,\ 2,\ \ldots,\ K$.

A graphical representation for the above code path is given in Fig. 3.

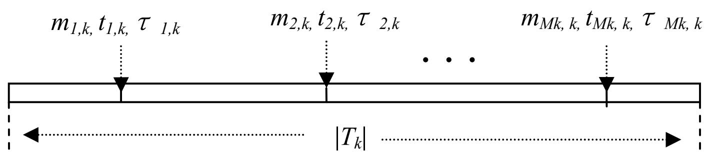
$m_{1,k},\ t_{1,k},\ \tau_{1,k}$
$m_{2,k},\ t_{2,k},\ \tau_{2,k}$
$m_{Mk,\ k},\ t_{Mk,\ k},\ \tau_{Mk,\ k}$
$|T_k|$

Fig. 3 $T_k(M_k;\ m_{1,k},\ t_{1,k},\tau_{1,k},\ \ldots,\ m_{Mk,\ k,},\ t_{Mk,\ k},\tau_{Mk,\ k})$

3.3 Event

Here an "event" is a loose term. It may refer to a specific code path mixture, a packet arrival process, a memory access, or a given instruction in a code path. Apparently, not all the events will have equal impact on the packet processing performance.

3.4 Design Methodology

3.4.1 Introduction

In this thesis, the key idea to the design of a fast, generic processor simulation tool is to decouple the common features pertaining to all possible processor architectures from the processor-specific features (e.g., I/O interface, bus, memory, and memory controller architectures).. Then focus on the modeling of the common features, while still being able to incorporate all the processor-specific effects that will impact the processor performance. On the basis of this idea, our approach is to focus on the modeling of non-processor specific components including core topology, multithreading, code path, code path mixtures, and packet arrival processes, pertaining to all the processor architectures, with plug-ins of a limited number of models that account for the processor-specific effects. These models, such as unloaded latency model and queuing model, are pre-developed and plug-in into the common part. Fig. 4 gives a logic diagram for the proposed methodology, which focuses on the modeling of three components:

(1) A packet arrival process;

(2) Code path association with arriving packets;

(3) A core that processes packets and produces performance data.

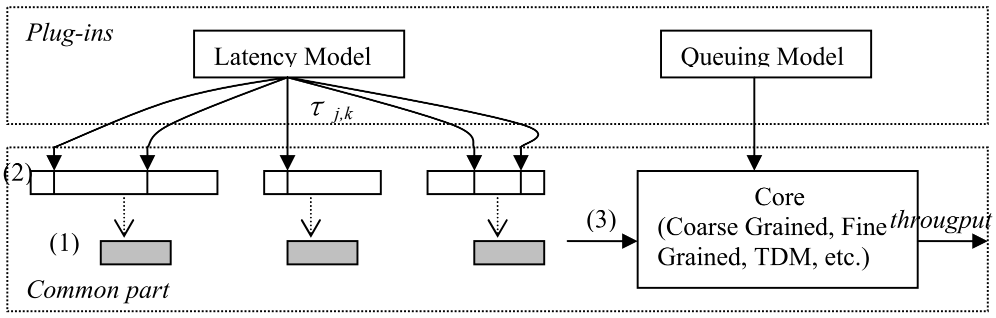

Fig. 4 Proposed Methodology

Each core is modeled at a highly abstract level, running any configurable number of threads handling various code paths based on a given thread scheduling discipline (e.g., the fine-grained, coarse-grained, or TDM-based discipline). This lightweight involvement of the processor-specific features is made possible for our methodology since our design goal is to focus on latency/throughput performance only, rather than rich features, such as program development. This allows us to focus on the design of the common part, independent of specific processor architectures. This also makes it possible to develop rich features easily, such as various thread scheduling disciplines.

3.4.2 Design Detail

The simulation software will be *event-driven*. The code path associated with an incoming packet will trigger a sequence of events handled by a given thread. Multiple threads, each handling a given code path, will then interact and compete for the core ALU and other supporting resources through a thread scheduling discipline. The tool will allow any instruction level events that have an impact at the thread level to be captured, such as average cost of context switching in the units of core cycles, dynamic code generation, serialization effect, critical section, etc.

To minimize the number of events to be processed in the core, *the simulator only capture the instruction level events that affect thread-level interactions.* This is based on the observation that interactions among threads are triggered by only a handful of events in the code paths and the semantics for the rest of instructions in the code paths are largely irrelevant. This observation allows the tool to simulate the thread-level events only, or instructions that causes, e.g., context switching or serialization effects, rather than all the instruction level details.

To see why this can be done, let's take a look at an example.

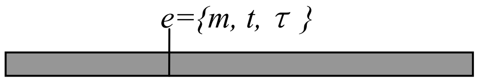

Fig. 5 An example of event-annotated code path

Consider an event-annotated code path in Fig. 5. In this code path, there is only one event *e*, which takes place at the *t*-th cycle for supporting component, *m,* access with unloaded latency $\boldsymbol{\tau}$. This event causes a context switching. Now consider two threads in a core, each handling a code path as in Fig. 5. They share the ALU resource based on a fine-grained thread scheduling discipline (i.e., switch context at every instruction). Fig. 6 gives the instruction execution timeline for the two code paths. The dark gray parts represent the code path segments. The light gray parts represent the cycles spent on event *e*, i.e., the loaded *m* access latencies. The white part stands for the cycles spent in the ready state waiting for execution after event *e* finishes. In this case, each code path involves three event boundaries: the start of the code path, the end of the code path, and the start and end of event *e*. The arrows represent the switches of control from one thread to the other after executing one instruction. The idea is to not to simulate each and every switch of control, but only the cycles at the event boundaries, i.e., the positions indicated by vertical lines. Since each code path may have up to a few dozens of events, only several dozens of event boundaries need to be simulated per packet. As a result, the event-driven simulation tool that captures only those events can run very fast, as our testing results showed (less than 10 seconds per simulation run on a Intel Duo core PC).

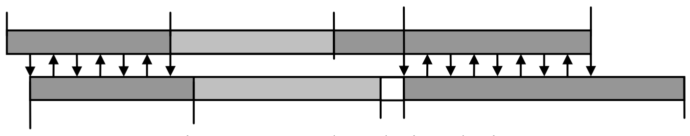
Fig. 6 Event-level simulation

In addition to *mem*, *Mem*, and *MEM*, there are a few other event types that need to be incorporated in a code path or pseudo code, such as instructions that trigger dynamic code generation, critical section and ordered processing. These events can be identified by the user and included in the pseudo code.

The above discussion indicates that it is possible to analyze the processor performance solely based on a pseudo code that can be built as soon as a mapping and the lengths (i.e., number of instructions or cycles) of the code segments between those events are given.

Finally, we note that there are two potential inaccuracies that may be introduced for not simulating instruction level details. They are: (1) the cost of the instruction pipeline aborts due to branching or context switching; and (2) instructions per core cycle in average sense only. One can expect that (2) will not introduce much performance error because the performance data is collected at a timescale much larger than per instruction timescale. Our testing based on IXP1200 simulator indicates that (1) may introduce about 1% to 5% throughput error. As part of the proposed research, we shall consider to compensate for this error by allowing the user to associate an average instruction pipeline abort cost with each context switching or branching.

3.5 Testing Results

In this section we test the accuracy of the simulator we developed based on the proposed ideas. The tool can simulate a single core with a coarse-grained thread scheduling discipline. We simulated a simple IPv4 Forwarding code sample available in IXP1200 mapped to the receive stage in IXP1200. The results are compared against the performance data obtained based on the Intel IXP cycle-accurate simulation. The code sample consists of a single code path which includes fourteen *mem* events in it and fifteen code path segments for packet processing (Please refer to Table A1 in Appendix for further detail). The IXP1200 cycle-accurate simulator (CAS) is run using infinite wire speed mode i.e. whenever a thread is not blocked and is polling port for a packet, and it will always get the packet immediately.

Table 1. The tool versus CAS forIPv4 Forwarding example

Thread	**Tool** Total Latency(TL)(cycles)	**CAS** Total Latency(TL)(cycles)	**Error %** rate |R1-R2|*100/R2
1	**483**	**486**	**0.62%**
2	**545**	**510**	**6.86%**
3	**593**	**555**	**6.85%**
4	**687**	**680**	**1.03%**

As shown in Table 1. The total latencies (TL) or data rates obtained by the tool are within 7% of CAS results. The results from the tool were produced within a few seconds with a single run, while CAS needed separate runs for each thread case with each run taking almost 30 minutes to stabilize, when the results were collected. The error is caused by not accounting for the processing overhead due to instruction aborts and the use of a simple FIFO queuing model for each resource access. The simulation results consistent with the results in Table 1 were also obtained based on a large number of other code samples available in the IXP1200/2400 simulators.

CHAPTER 4

STUDY OF CACHING PERFRORMANCE

This chapter describes the simulation results on the effect of caching IP forwarding table entries for packet processing based on a single core multithreaded processor. We performed simulation using the simulator we developed. The simulation is designed to emulate a typical IP packet forwarding using an IXP1200 NP with the addition of caching for IP-lookup and characterizes the benefits and drawbacks of caching as a potential mechanism to improve real-time packet processing performance. The caching performance is tested in a large parameter space in terms of miss ratio, number of threads, and line rate, and under various delay/loss constraints.

4.1 Simulation Setup

In this study, we use the same IPv4 Forwarding code path as the one used in Chapter 3 (see Table A1 in Appendix). To account for the caching effect, we added a new *event,* called *cache* event. Now, there are fifteen segments in the code path, including fourteen *mem* type events and one *cache* event. If the cache hits, the processor will perform cache only lookup without memory accesses for IP-lookup, otherwise, it will incur a miss penalty including two cycles for cache access and the four memory accesses and the related four code segments execution for IP-lookup (see Table A1 for details). The cache miss penalty includes 59 instructions and 2 cycles of cache access latency. In the simulation, we assume that different packets will have equal probability to incur a cache miss and there is no correlation among different cache misses. .

We assume the processor ALU clock rate is 1 GHz and the packet size is 64 Bytes for all packets. To test caching effect under given line rate, we let packets come into the processor at fixed interval Tp (in the units of ALU cycles). The line rate R = 64x8/Tp (Gbps). So, by changing Tp, we effectively change line rate R.

To account for the threading effect, we tested various numbers of threads in the range of 1 to 8. To test the processing performance under various delay/loss constraints, we assume that the processor has an input buffer of three different sizes, i.e., 2, 4 and 8 packet sizes. In practice, the available input buffer size varies from one NP to another. But in general, it is small and on the same order as the number of configurable threads. For some NPs based on cut-through switching architecture, there is virtually no input buffering and the packets are processed on-the-fly. For this very reason, we consider the cases from 1 all the way to 8, the maximum number of threads considered in this study.

We also tested the throughput under three different miss ratios. They are 0.1% as in [16] which is the lowest miss ratio reported, 27.2% as in [15] which is the highest one in IP lookup caching, and 10% as reported in [20,21,22].

In the simulation, we generated one million packets with the code path we just defined. For fixed miss ratio, number of threads, and size of buffer, we increase the input line rate to find the maximum line rate without packet drops.

4.2 Simulation Results and Analysis

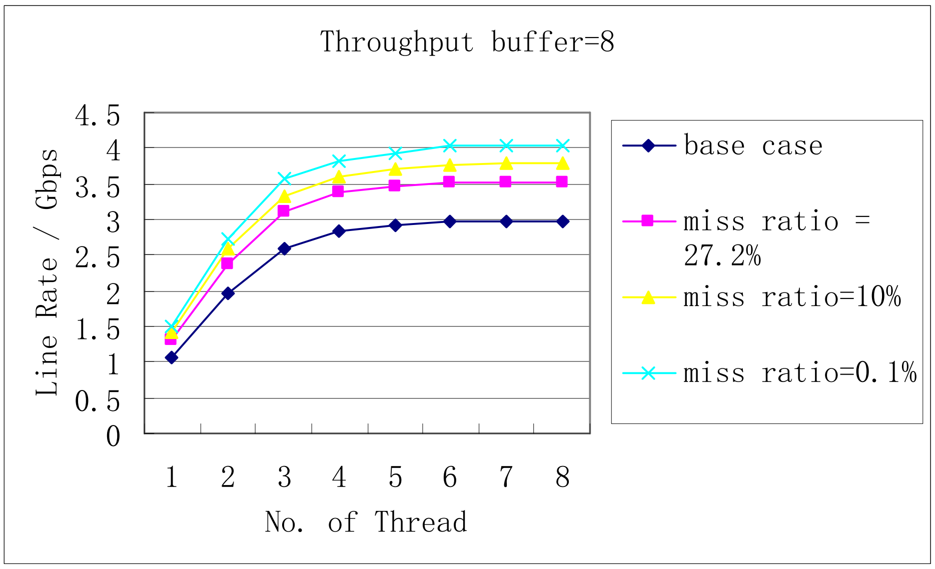

Figure 7. The throughput under different miss ratio when input buffer size is 8-packets

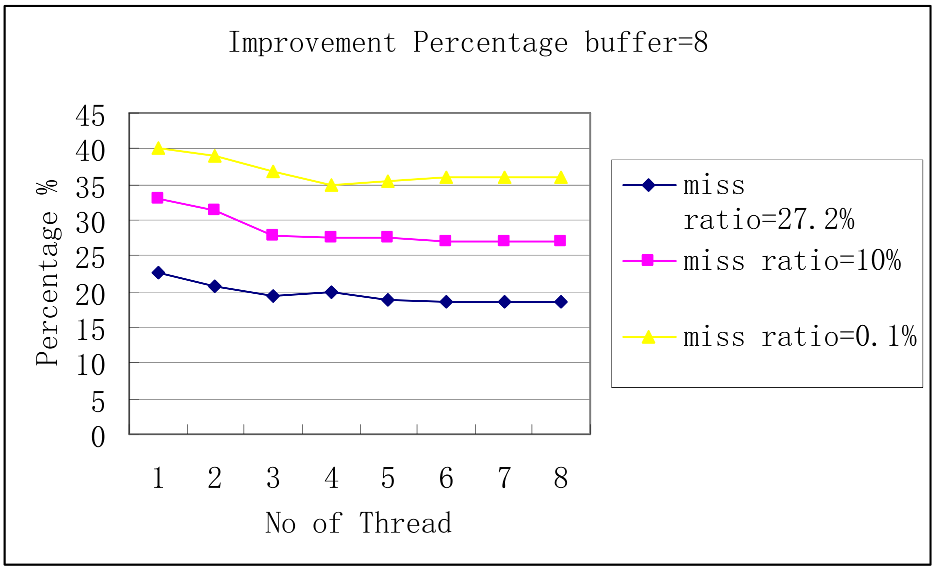

Figure 8. The percentage of throughput improvement under different miss ratios when input buffer size is 8-packets

Figure 7 shows the throughput when input buffer is 8-packets size. The base case is the maximum throughput of processor under different number of threads without caching. Clearly, a smaller miss ratio can achieve higher throughput performance. For miss ratio equals 0.1%, the processor can support the highest line rate, i.e., 4.02 Gbps. From figure 8, we see that the percentage of throughput improvement decreases rapidly, for example, at miss ratio = 10%, the percentage of throughput improvement reduces from 33% to 27% as the number of threads increases from 1 to 6. For other two curves, the percentage also decreases monotonously as the number of threads increases. The maximum sustainable line rate saturates when the number of threads exceeds 6. This is because for this particular code path, 6 threads can already completely hide all the memory access latencies. So adding more threads can not help increase the throughput. This point is also supported by Figure 7, in which the throughput of baseline doesn't improve by adding more than 6 threads. From these two figures, we conclude that the benefit of caching becomes weakened as the number of threads increases.

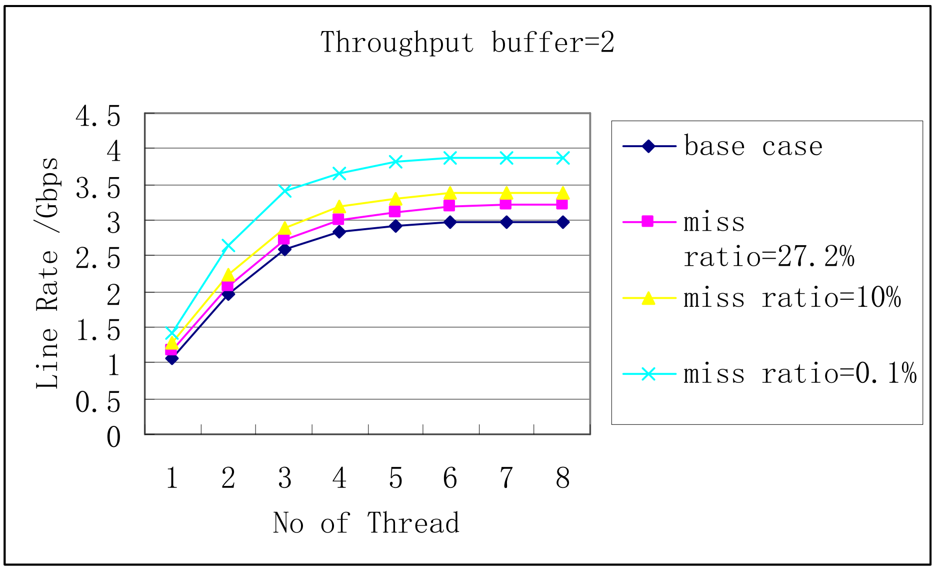

Figure 9. The throughput under different miss ratio when input buffer size is 2-packets

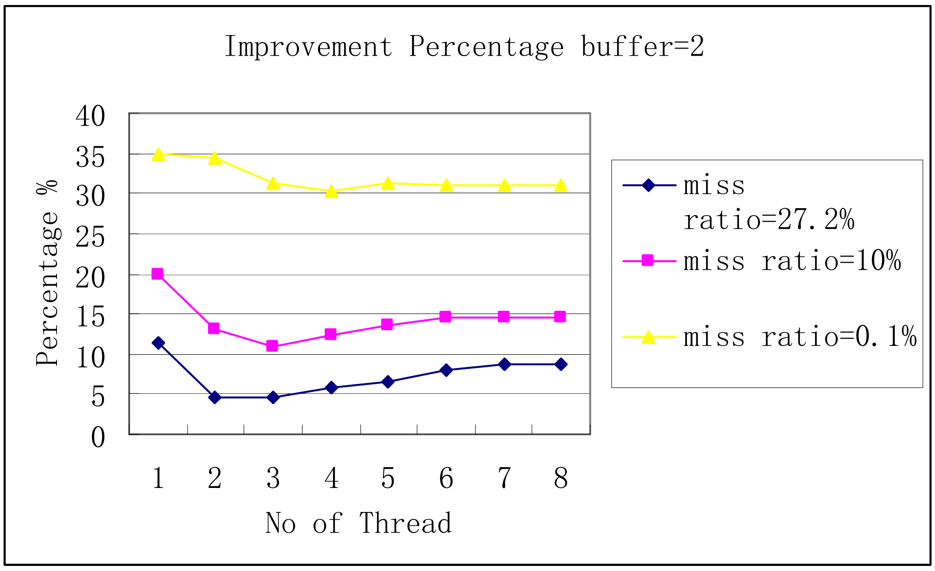

Figure 10. The percentage of throughput improvement under different miss ratio when input buffer size is 8-packets

Figure 9 shows the throughput when input buffer is 2-packets size. Comparing to Figure 7, since the drop/loss constraints (i.e. the input buffer size) is more stringent, the throughput is relatively lower, especially when miss ratio is big. Comparing the two curves of miss ratio = 27.2% in Figure 7 and 9, we find that the curve in Figure 9 is much closer to the baseline case. From figure 10 we see that the percentage of throughput improvement is 4.6% when the miss ratio is 27.2%. This means the caching technique is still helpful even if the drop/loss constraint is very tight.

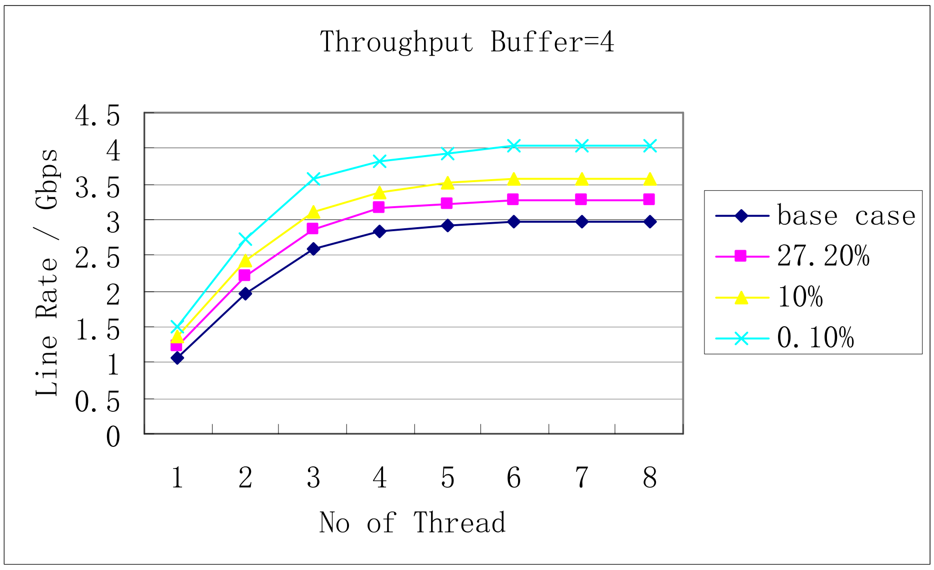

Figure 11. The throughput under different miss ratio when input buffer size is 4-packets

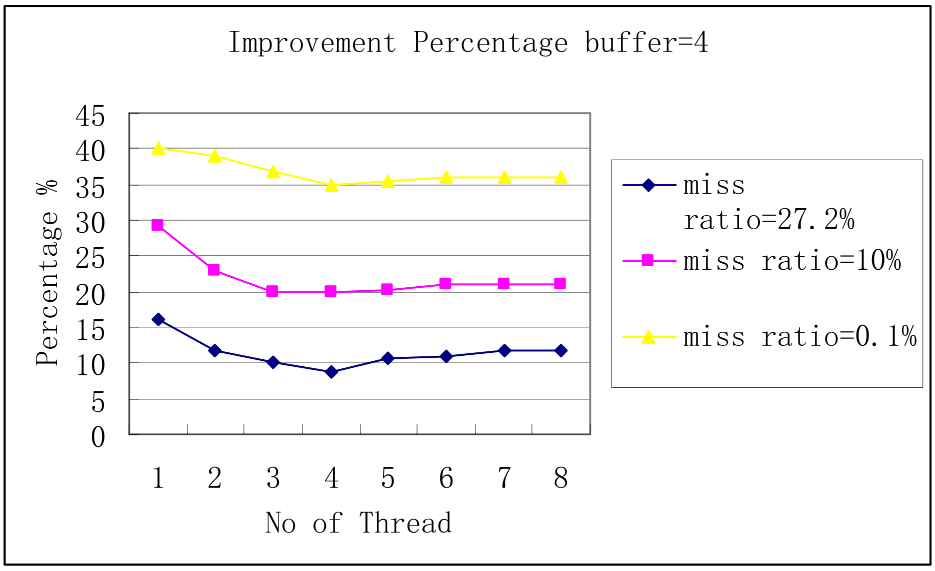

Figure 12. The percentage of throughput improvement under different miss ratio when input buffer size is 4-packets

Figure 11 shows the throughput when input buffer is 4-packets size and Figure 12 shows the percentage of throughput improvement. Comparing the three curves of percentage at miss ratio = 27% in Figure 8, 10 and 12, we find that in Figure 8 the improvement percentage decreases monotonously when the number of thread increasing. In Figure 10 and 12, however, the improvement percentage first decreases until the number of threads reaches 4 and 2, respectively. Then, the improvement percentage increases and fattened out at 7 and 8, respectively. This observation indicates that there are three factors will work together to affect the caching effect. They are number of continuous cache misses, input buffer size and number of threads. Obviously, increasing the number of continuous cache misses (i.e. increasing the miss ratio) will reduce the caching effect; both input buffer and multi-threading can help hide the continuous cache misses, so increasing input buffer size and using multi-threading can help to increase the caching effect; increasing the number of threads itself does help hide memory access latencies and reduce the caching effect. We can verify these points from these figures. In Figure 8, since the buffer size is big enough to hide the continuous cache misses, increasing the number of threads does help hide memory access latencies and reduce the caching effect. More over, we can find that increasing the number of threads will reduce the caching effect more effective when thread number is low. In contrast, in the case of relatively small buffer sizes, as in figure 10 and 12, adding threads will reduce the caching effect a lot when number of threads is small, so the percentage of improvement decreases at first. However, since buffer size is small, buffering cannot hide all the continuous cache misses. As a result, multithreading can help hide the miss effect, that is the reason of the improvement percentage will increase when we add more threads. Later, after we added enough threads to hide the cache miss effect, if we add more threads, these threads will help to hide more memory access time during the processing and the improvement percentage can not increase. We also can verify this point by comparing the curves at miss ratio = 10%. The curves at miss ratio = 0.1% almost decrease monotonously since the miss ratio is too low, and even a 2-packets size buffer can hide the entire continuous cache misses effect.

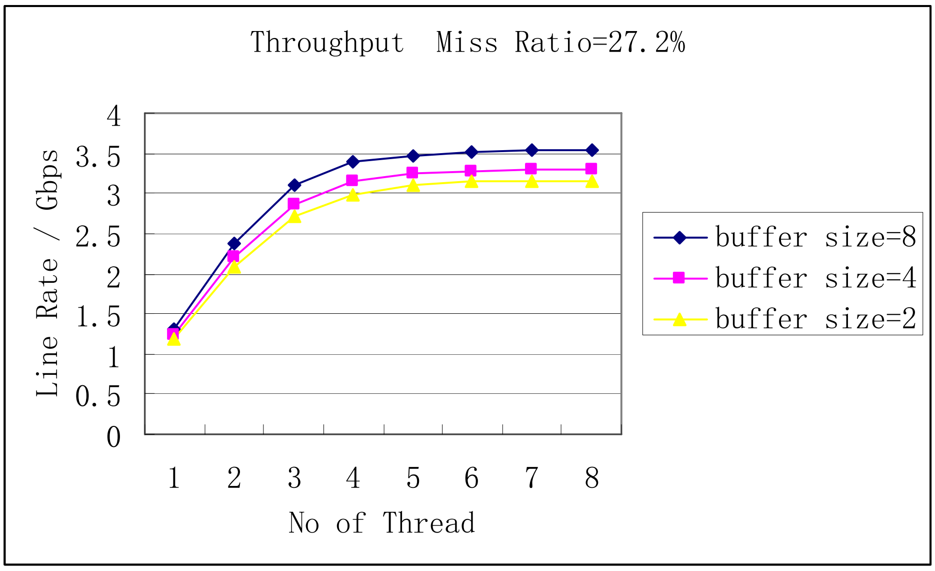

Figure 13. The throughput under different size of input buffer when miss ratio is 27%

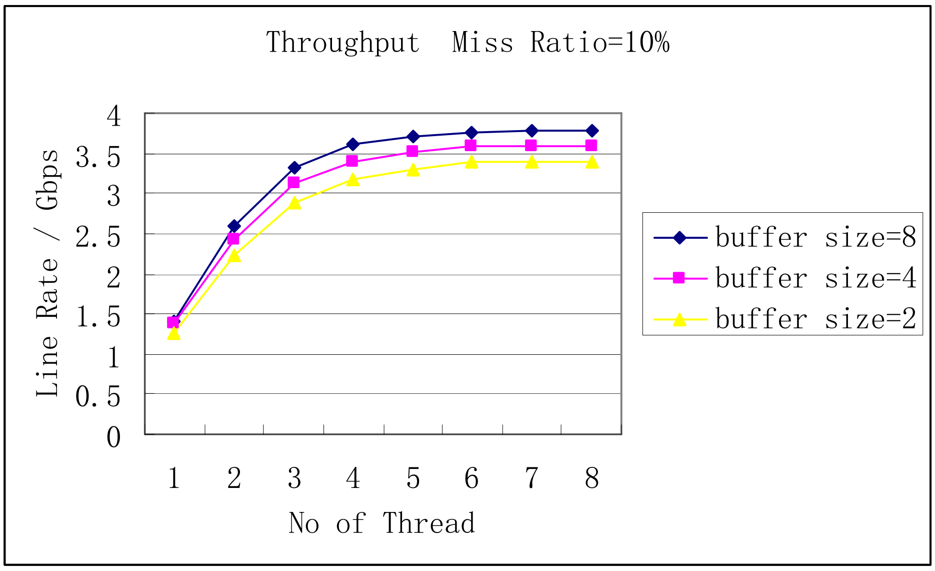

Figure 14. The throughput under different size of input buffer when miss ratio is 10%

Figure 13 and 14 shows the throughput at different miss ratio = 27.2% and 10% respectively. From Figure 13 and 14, we note that looser drop/loss constraints can improve caching performance since the curve with larger buffer size is always above the one with smaller size. Comparing Figure 13 and 14, we find that the two curves for buffer size 8 and 4 are closer in Figure 14 than in Figure 13. This indicates that increasing buffer size can be more efficient when miss ratio is large. It is easier to see this point by looking at Figure 15. On average, 8-packets size buffer can improve throughput by 7.2% when miss ratio is 27.2% compare to 5.5% when miss ratio is 10%.

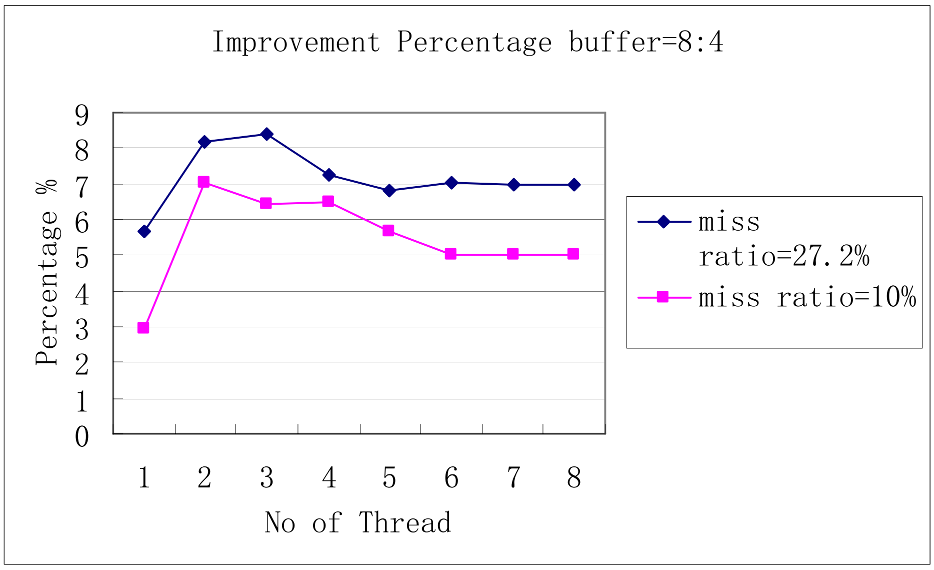

Figure 15. The percentage of throughput improvement when buffer size change from 4 to 8 under miss ratio is 27.2% and 10%

In summary, the above studies indicate:

(1) When the cache miss follows a uniform distribution, the caching technique is always helpful to increase the throughput performance. Even if the miss rate is big, (i.e. 27.2%), and the delay/loss constraints is relatively tight (i.e. 2-packets input buffer size), the caching can help achieve at least 4.6% throughput improvement comparing with the baseline case.

(2) Both input buffer and multithreading can help processor hide the continuous cache misses. The caching benefit will be weakened when delay/loss constraint is tight and the number of threads is small.
(3) When the drop/loss constraint is loose, the use of a large number of threads can effectively hide the memory latency, making caching less effective.
(4) Caching can be effective in the parameter range where the delay/loss constraints are loose (i.e. the input buffer size is large), especially when miss ratio is big.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, we first build a lightweight processor simulator, which can be easily adapted to any multithreaded, single-core processor architecture. We then tested simulator using the IPv4 Forwarding code path provided by the IXP1200 cycle-accurate simulator and compared the results with IXP1200 cycle accurate simulation. Our simulator can provide packet throughput/latency data within 10% of the IXP1200 results. Moreover, this simulator is extremely fast and requires only a pseudo code as input for simulation (note that cycle-accurate simulators require micro-code as input for simulation). As a result, it allows effective performance analysis of a multi-threaded, single-core NP/processor in a large parameter space.

Second, we apply this simulator to the study of the effect of caching for IP-lookup on the packet throughput performance. The simulator emulates a typical IP packet forwarding using an IXP1200 NP with the addition of caching for IP-look. The simulation study is performed in a large parameter space. The caching technique is found to be always helpful in improving the throughput performance when the cache miss distribution is uniform. More over, caching effect will be weakened when delay/loss constraint is tight and the number of thread is small.

5.2 Future Work

In this thesis we have studied only uniform cache miss distribution. Different input streams, different cache configuration, hashing functions and update algorithms can generate different cache miss distributions and can affect the effectiveness of caching. As part of our future work, we shall study the impact of other cache miss distributions on the performance of caching.

In this thesis, we only developed the simulator for multi-threaded, single-core processor. We shall extend our simulator to emulate multi-core processor and then use this simulator for the study of the impact of multi-level caching techniques on real-time packet processing.

APPENDIX A
CODE PATH USED IN SIMULATION

Table A1 Code path for **IPv4 Forwarding** (IXP 1200)

Task	**# instructions in code segment ($t_{m,k} - t_{m-1,k}$)**	**Type of I/O access**	**Unloaded latency** $\tau_{j,k}$
Check receive ready flags	5	FBI read	14
Move packet from IX Bus to RFIFO	8	FBI write + IX Bus receive	76
Read recive control information (after reading packet from IX Bus to RFIFO)	2	FBI read	19
Wait for buffer allocation (in SDRAM); get the descriptor from SRAM	11	SRAM read	17
Read 3 Quad words from RFIFO into microengine for IP validation	16	RFIFO read	18
Read 2nd 32 byte to SDRAM (in the allocated buffer)	15	RFIFO read	22
IP lookup	40	SRAM read	17
IP lookup	7	SRAM read	17
IP lookup	5	SRAM read	17
Get next hop information from SDRAM	7	SDRAM read	47
Write packet descriptor to SRAM (after associating it with a TX port)	16	SRAM write	18
Read queue descriptor from SRAM (for enqueue operation)	4	SRAM read	22
Write the packet descriptor to SRAM (to the TX queues associated with the TX port)	15	SRAM write	20
Miscellaneous	6		
TOTAL	**157**		**324**

REFERENCES

[1] Hao Che, C. Kumar, and B. Menasinahal, " Fundamental Network Processor Performance Bounds," in *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA05)*, August 2005.

[2] Crowley, Marc E. Fiuczynski, jean-Loup Baer, “On the Performance of Multithreaded Architectures for Network Processors”, Technical Report 2000-10-01, Department of computer Science & Engineering, University of Washington, Seattle, WA 98195.

[3] Wajdi Feghali, Brad Burres, Gilbert wolrich, Douglas Carrigan “Security: Adding Protection to the Network via the Network Processor”, Intel Technology Journal, Volume 6, Issue 3, 2002.

[4] Ram Bhamidipati, Ahmad Zaidi, Siva Makineni, Kah K. Low, Robert Chen, Kin-Yip Liu, Jack Dahlgren, “Challenges and Methodologies for Implementing High-Performance Network Processors”, Intel Technology Journal, Volume 6, Issue 3, 2002.

[5] Haiyong Xie, Li Zhou, Laxmi Bhuyan, “Architectural Analysis of Cryptographic Applications for Network Processors”, Department of Computer Science &Engineering, University of California, Riverside, Riverside.

[6] R. Ramaswamy, N. Weng, and T. Wolf, “Analysis of Network Processing Workloads,” in *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 226-235, Austin, TX, March 2005.

[7] Patrick Crowley, Jean-Loup Baer, “A Modeling Framework for Network Processor Systems”, In *Proceedings of the HPCA-8 Workshop on Network Processors*, 2002.

[8] Niraj Shah, William Plishker, Kurt Keutzer “NP-click: A Programming Model for the Intel IXP1200” 2nd Workshop on Network Processors (NP-2) at the 9th International Symposium on High Performance Computer Architecture (HPCA-9), Anaheim, CA, February, 2003.

[9] Vik Chandra, “Selecting a network processor”, IBM Microelectronics

[10] L. Thiele, S. Chakraborty, M. Gries, S. Kiinzli, “Design Space Exploration of Network Processor Architectures,” *Network Processor Design: Issues and Practices*, Editors: P. Crowley, M. Franklin, H. Hadimioglu, P. Onufryk, Morgan Kaufmann Publishers, October 2002.

[11] L. Thiele, S. Chakraborty, M. Gries, S. Kiinzli, “A Framework for evaluating Design Tradeoffs in Packet Processing Architectures” *Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology (ETH) Z urich, CH-8092 Z urich, Switzerland.*

[12] Matthias Gries, Chidamber Kulkarni, Christian Sauer, Kurt Keutzer, “Comparing Analytical Modeling with Simulation for Network Processor: A Case Study”, Design, Automation and Test in Europe Conference and Exhibition, 2003, Volume , Issue , 2003 Page(s): 256 - 261

[13] Prashant R. Chandra, Frank Hady, Raj Yavatkar, Tony Bock, Mason Cabot and Philip Mathew “Benchmarking Network Processors” Intel Corporation.

[14] Keith Morris, "Challenges in Making Highly Integrated Network Processors", Applied Micro Circuits Corporation

[15] Kartik Gopalan, Tzi-cher Chiueh "Improving Route Lookup Performance Using Network Processor Cache", In Proc. of SC2002 High Performance Networking and Computing, Baltimore, MD, November 2002,

[16] Huau Liu "Routing Prefix Cacheing in Network Processor Design", Tenth International Conference on Computer Communications and Networks, Proceedings 2001 Volume , Issue , 2001 Page(s):18 - 23

[17] Tzi-cher Chiueh, Prashant Pradhan "Cache Memory design for Network Processors", Proc. High Performance Computer Architecture, pp. 409-418, 1999.

[18] Kang Li, Francis Chang, Damien Berger, Wu-chang Feng "Architectures for Packet Classification Caching". The 11th IEEE International Conference on Networks, 2003. ICON2003, Volume , Issue , 28 Sept.-1 Oct. 2003 Page(s): 111 - 117

[19] Francis Chang, Wu-chang Feng, Kang Li "Approximate Caches for Packet Classification", INFOCOM 2004, Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, Volume 4, Issue , 7-11 March 2004 Page(s): 2196 - 2207 vol.4

[20] C. Partridge, "Locality and Route Caches", Position Statement inNSF Workshop onIternet Statistics Measurement and Analysis, San Diego, Ca, USA, Feb. 1996.

[21] J Xu, M. Sinhal, and J. Degroat, "Anovel cache architecture to support layer-four packet classification at memory access speeds" in Proc. of INFOCOM 2000, tel

Aviv, Israel, pp. 1445-1454, March 2000.

[22] B.Talbot, T. Sherwood, and B. Lin, "IP Caching for Terabit Speed Routers", Globlecom, 1999

[23] P. Newman, G. Minshall, T. Lyon, and L. Huston, "IP switching and gigabit routers", IEEE Commun. Mag., Bol. 35, pp. 64-69, Jan 1997.

[24] Passive Measurement and Analysis project, Nationsl Laboratory for Applied Network Research. http://moat.nlanr.net/pma

BIOGRAPHICAL INFORMATION

Miao Ju received his Bachelor of Science degree in Information and Computational Science from BeiHang University, Beijing, China in 2005. He began his graduate studies in the Department of Computer Science and Engineering at The University of Texas at Arlington in Spring 2006 and received his Master of Science degree in August 2007.