網路處理器資源分配策略：針對記憶體存取密集的應用程式

學生 ： 張耀中　　指導教授 :林盈達

國立交通大學資訊科學系

摘要

今日的網路應用，如入侵偵測系統(NIDS)，需要的大量記憶體存取已超過多用途處理器所能容忍的程度。網路處理器是有別於特定功能積體電路的另一選擇。其多微引擎提供平行處理的能力，且多執行緒隱藏記憶體存取需要的時間。然而，效能取決於適當的微引擎和執行緒配置。本論文利用 IXP2400 網路處理器來實作一個入侵偵測系統，並且探討不同資源配置對於效能的影響。本論文的價值為下列結論。第一，在微引擎使用率不超過 100%的情況下，給定一個應用程式，執行緒的總數(即 $I \times J$ ，I, J 各代表微引擎數量和每個微引擎內執行緒數量)影響系統的吞吐量。第二，$I \times J$ 不斷的增長會使得 SRAM 的使用率超過 k，k 為有效率使用記憶體的上限。第三，給定一個應用程式、演算法和 k，可以推導一組最佳的(I, J)。第四，當 SRAM 變成瓶頸時，即 $I \times J > k$ ，多重記憶體單元可以用來解決此問題。

關鍵字: 網路處理器，記憶體存取密集，微引擎，執行緒，入侵偵測系統

Allocation Strategies of Network Processors for Memory Access Intensive Applications

Student: Yao-Chung Chang Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science

National Chiao Tung University

Abstract

Networking applications nowadays, such as network intrusion detection system (NIDS), require more memory accesses than the original processor can tolerate, where the network processor architecture is an alternative solution different to ASIC. The architecture provides parallelism through multiple microengines (MEs) and hides memory-access latency through hardware threads. However, its performance depends on proper ME, thread allocations. In this work, we develop an NIDS over the Intel IXP2400 and investigate the impact of resource allocation on its performance. Our paper is rich for the following conclusions. First, given an application and algorithm, the throughput is influenced mostly by the total number of threads, namely $I \times J$, where I and J referred to as the number of ME and threads per ME, respectively, as long as the ME utilizations do not exceed 100%. Second, the bottleneck is found to be the SRAM as $I \times J$ expands and exceeds the upperbound, k, which *cost-effectively* utilizes the memory. Third, supposed an application, algorithm and k, an optimal (I, J) can always be derived. Fourth, multiple memory banks can be adopted to tackle the SRAM bottleneck, namely when $I \times J > k$.

Keywords: Network Processor, memory-access intensive, microengine, thread, N IDS

Contents

List of Figures

List of Tables

Chapter 1. Introduction

Networking applications nowadays that offer extra security and content-aware processing demand for more powerful hardware devices to achieve high performance. For memory-access intensive applications, such as Network Intrusion Detection Systems (NIDSs) [1], general purpose processors with high speed memory banks are often adopted; however, the cost is very considerable while the throughput is not satisfactory for that the processor's utilization is low because of much memory-access overhead.

Rather, the Application-Specific Integrated Circuits (ASICs) [2] can meet the performance requirement with a circuitry designed for strict guarantees on memory-access latency using pipelined architecture and embedded memory. Nonetheless, the lack of flexibility and long period of development make it less appealing.

Network processors [3] are emerging to be an alternative to the above-mentioned problems for their multithreaded multiprocessor architecture, flexibility and shortened development cycle. Multiple processors allow simultaneous data-plane processing of multiple packets on a cluster of processors. Moreover, the hardware threads having very little context switch overhead can hide the memory-access latency [4]. Thus, the nature of parallelism and latency hiding can greatly increase the throughput of packet processing. Besides, network processors offer the flexibility through its re-programmability, making functional adaptations much easier than ASIC, which otherwise needs to be re-designed.

Since the hardware resources of network processors are richer than these of general purpose processors, the resource allocation, such as processors, threads and

memory is critical to the performance. For memory-access intensive applications, some related researches have been devoted to improve the throughput by the deployment of network processors. Bos and Huang [5] implemented an NIDS over the Intel IXP1200 [6]. The prototype comprises only the receiver and packet processing using Aho-Corasick [7] algorithm, but it does not support inspection of patterns across more than two packets as well as multiple flows. Clark, et al, [8] designed a Network Intrusion Detection and Prevention System (NIDP) utilizing an IXP1200 and an FPGA performing the matching for packet header and payload respectively, in which bottleneck is found to be the bus connecting them. Nevertheless, none of both addressed the impact of allocations of processors, threads, and memory banks on performance which influences the throughput and utilizations of processor and memory bank.

In this work, we explored the feasibility of implementing a memory-access intensive application, namely an NIDS, over the Intel IXP2400 [9], which has a set of characteristics common to most network processors. Several software components referred to as processing stages [10] were designed for packet reception and transmission, classification, thread dispatcher and signature matching. Among all string matching algorithms, the two, Aho-Corasick and Wu-Manber [11], were commonly adopted for signature matching due to that they are easy for implementation as well as popular in most network applications, for example, Snort. We use an intuitive allocation of processors and threads for each processing stage, aiming to suggest a possible adjustment according to the results of both external and internal benchmarks. The former characterized the throughput figures of the implementation, while the latter carried out some in-depth analysis of the memory, processor utilization with regards to the allocations of processors, threads, and memory banks when different algorithms were used. Some questions were

investigated and discussed including: 1) Task allocation and bottleneck observation, 2) Effect of improper ME/thread allocations, 3) Optimal numbers of MEs and threads within each ME, 4) Effectiveness of employing several memory banks. Our preliminary results show that increasing the total number of threads improves the throughput, besides the performance benefits from multiple memory banks considerably.

This paper is organized as follows. Chapter 2 describes the hardware architectures of IXP2400. Chapter 3 describes the problem statement. Chapter 4 elaborates the design and implementation of our system. Chapter 5 presents the results and observations in the external and internal benchmarks. Some conclusive remarks of this article are made in chapter 6.

Chapter 2. Hardware Architecture of IXP2400

2.1 Hardware Architecture of IXP2400

As depicted in Fig. 1, IXP2400 consists of several components connected by a bus. The design of the hardware is categorized into the following features.

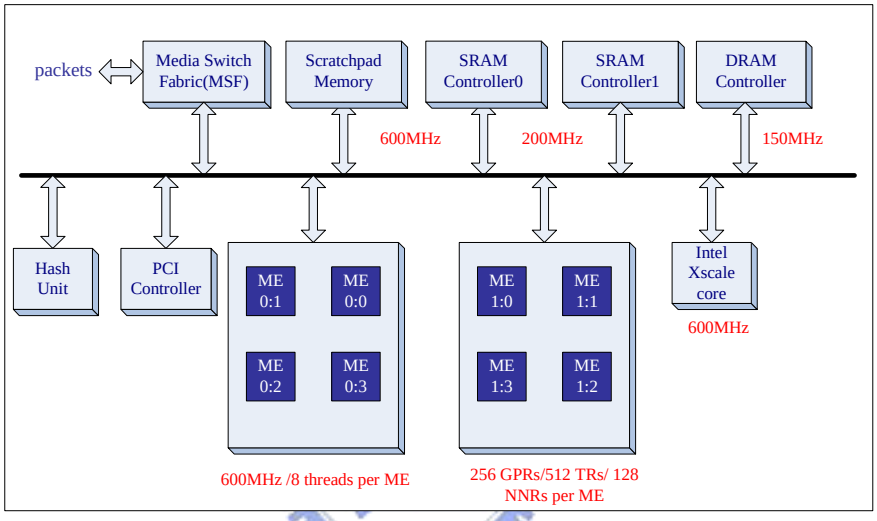

Fig. 1. Hardware architecture of IXP2400

Multithreaded multiprocessor architecture

The IXP2400 features nine programmable processors: one Intel XScale core [12] and eight microengines (MEs), operating at 600MHz. The Intel XScale core is responsible for housekeeping functions such as table initialization and exception processing for control-plane packets such as ICMP unreachable packets. Data-plane processing, which accounts for the most part in packet processing, is implemented on MEs. Each ME has eight hardware threads, each of which having its own register set and program counter to support fast context switch when memory accesses occur.

Versatile memory hierarchy

To ease memory-access overhead, IXP2400 exploits four different memory types,

DRAM, SRAM, scratchpad, and local memory in an ME, given tradeoffs between size and latency. IXP2400 has one channel of DDR running at 300MHz. The channel can support up to 2GB of DRAM, yielding enough capacity for storing packet buffers. In addition to DRAM, IXP2400 also provides two channels of Quad Data Rate (QDR) SRAM running at 400MHz. Up to 16MB of SRAM can be populated on each channel The SRAM is primary for accommodating packet descriptors, queue descriptors and other data structures frequently used. Furthermore, the on-chip 16KB scratchpad memory running at 700MHz provides very similar capability to SRAM. In the rest, local memory inside each ME is 2560 words in size and often used as a cache for smaller data structures.

Flexible external interface

The Media Switch Fabric (MSF) is an external interface used to connect the Intel IXP2400 to a physical layer device and/or a switch fabric. The MSF consists of receiving and transmitting interfaces which can be configured for different protocols such as POS PHY Level 3 [13] and CSIX-L1 [14]. Incoming packets are received into the Receive Buffer (RBUF) and outgoing packets are held in the Transmit Buffer (TBUF), which are both 8KB in size. The MEs can move data from RBUF to DRAM and from DRAM to TBUF using the DRAM[rbuf_rd] and DRAM[tbuf_wr] instructions directly, greatly avoiding packet duplications and unnecessary memory accesses.

Coprocessors

Two kinds of hardware coprocessors, including a hash unit shared by all MEs and a Cyclic Redundancy Code (CRC) unit inside each ME, are incorporated in the system. The hash unit is capable of 48-bit, 64-bit and 128-bit polynomial divisions.

Performing a high-quality hash in software is cycle-consuming, which occurs frequently in packet classification, and thus it should be offloaded to the coprocessor. Besides, a high quality hash will uniformly distribute entries in the smaller table to reduce the probability of a hash collision, and therefore resulting in fewer memory accesses. In addition to the hash unit, each ME contains a CRC unit providing a similar functionality to the hash unit for offloading CRC computation.

2.2 Detailed Packet Flow in IXP2400

The processing flow of an ordinary packet is elaborated below referring to Fig 1. Upon the arrival of a packet at the MSF of IXP2400, the MSF partitions the packet into several smaller chunks called mpackets, which can be configured to 64, 128, and 256 bytes in size, and places them into the RBUF elements. The threads of the MEs dedicated for packet receiving in turn perform the reassembly of mpackets, and move them directly from the RBUF into DRAM, in which MEs and the Intel XScale core carry out further operations. The packet processing typically consists of packet classification followed by packet modification. During packet processing at MEs, chances are that some exception handling and housekeeping are manipulated by the Intel XScale core through the interrupt and message queue mechanism. In the later scenario of packet flow, the transmission process is just the reverse of the reception process, namely the packet is segmented into several mpackets by the threads dedicated for packet transmission, and then placed into the TBUF.

Chapter 3. Problem Statements

In this paper, we focus on the impact on performance by the processor, thread and memory bank allocations when implementing memory-access intensive applications on the Intel IXP2400 network processor. Some problem statements are discussed below.

(1) Task Allocation and Bottleneck Observation

Before implementing an NIDS, some functional blocks referred to as *processing stages* need to be identified and then mapped to the platform. During the mapping process, we try to properly exploit the hardware features such as hierarchical memory structure and multithreaded multiprocessor architecture. This mainly involves the assignment of memories to store different data structures, as well as the allocation of threads and MEs. The possible bottlenecks will be identified after the system is implemented.

(2) Effect of Improper ME/Thread Allocations

The performance of an application is affected by two factors, the computing power and memory-access latency. The former is determined by the number of processors used referred to as I, while the latter can be alleviated by adjusting the total number of threads employed, namely $I \times J$ [15]. Observing that total numbers of processors and threads are fixed to the hardware platform, it is interesting to see how an allocation (I, J), especially an improper one, affects the system performance.

(3) Optimal Numbers of MEs and Threads within Each ME

It is known that memory-access intensive applications benefit from increasing the total number of threads, namely $I \times J$ rather than individual *I* and *J*, because of its ability of hiding memory-access latency. Nonetheless, how to determine a fitting $I \times J$, given a certain hardware spec such as clock rate and memory service rate, remains unanswered. In addition, we are also interested in finding a optimal (*I*, *J*) combination, regardless of the limit on the numbers of MEs and threads per ME of the platform. A (*I*, *J*) is considered optimal when the utilizations of both ME and memory are *cost-effectively* high, as will be explained in chapter 5.

(4) Effectiveness of Employing Several Memory Banks

Multiple memory banks reduce the average memory access latency. For memory-access intensive applications, more memory banks are supposed to improve the performance. Nonetheless, the effectiveness could be influenced by whether the accesses are evenly distributed into memory banks. Some experiments are therefore designed to investigate the feasibility of adding memory banks for memory-access applications.

Chapter 4. Design and Implementation

In this chapter, we first introduce basic operations of an NIDS, and then characterize its processing stages in order to map them onto IXP2400. Finally the design and implementation of an NIDS over IXP2400 is described.

4.1 NIDS Briefing

The processing of an NIDS, for example, Snort [1], mainly consists of three phases: a packet decoding phase which sets up pointers to packet data at different layers and stores them into data structures for later analysis by the detection engine; a detecting phase, in which a group of rules matched by a packet header are applied for further signature matching, and an alert phase, in which some alert or logging routines are carried out. Although later versions of Snort include the preprocessing phase performing the IP de-fragmentation and TCP stream reassembly, it is optional and can be turned off. Among these phases, recent measurement of Snort [16] on a production network shows that at least 31% of total processing time is consumed by the detecting phase, while the rest is spent mostly on disk I/O.

4.2 Design Issues

According to the above-mentioned characteristic of an NIDS, it is clear that we can implement an NIDS over IXP2400 by dividing the packet processing into a series of stages, namely receiver, packet inspector and transmitter, and mapping them onto the MEs. We do not consider the preprocessing stage since oftentimes it is not done in the fast path [17], but by the XScale. Moreover, packets can be distributed to a pool of

MEs, and thus threads, in the packet inspector to exploit high parallelism. Nevertheless, two problems including *packet ordering* and *flow interleaving* arise.

Packet ordering

The issue of packet ordering happens in a processing stage when multiple threads are dispatched to process the corresponding packet simultaneously. If the amount of time to process a packet is not constant, the packet ordering is no longer guaranteed as shown in Fig. 2a. To tackle this problem, a mechanism named *ordered threads* [18], is adopted to keep packets in strict order, meaning that threads in a processing stage consisting of several functions handle packets in order as presented in Fig. 2b. The synchronization among threads is simple since only one thread is allowed to enter a function, for example, function 2 in Fig. 2b, which may access a global variable at a certain instant. The ordering is supported in a very efficient manner using a feature of the hardware called inter-thread signaling. However, this mechanism suffers from the performance degradation due to the poor parallelism and less hiding of memory-access latency [18].

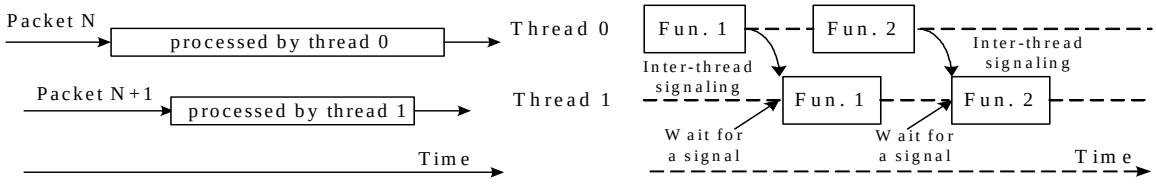

Fig. 2. Timeline showing two consecutive packets (a) being out of order, and (b) being ordered in a processing stage

Flow interleaving

A pattern may stretch across multiple packets when doing packet inspection. If flows are interleaved, it is not guaranteed that two consecutively processed packets belong to the same flow; in this way patterns across multiple packets can not be inspected appropriately.

For such cases, we refine our design by adding two processing stages with strict packet ordering, namely flow classifier and thread dispatcher. The main idea behind is to classify packets into different flow queues such that flows are no longer interleaved. Further, each thread in the packet inspector is dispatched by a *dispatcher* to serve one flow queue. After finishing the inspection of a packet, the packet inspector thread stores the final state of inspection for later reference by another thread serving the same queue. So, patterns across multiple packets can be inspected and the packet ordering in the same flow is maintained.

4.3 Mapping Processing Stages to the Hardware Platform

Fig. 3 shows the processing stages of an NIDS, its corresponding task and resource allocation on IXP2400. The NIDS processing is elaborated as follows. On receiving a packet from an input port, the packet data is moved from RBUF to DRAM; the corresponding packet descriptor is stored in SRAM and also passed to the next stage through the receiving scratch ring. Subsequent stage, the flow classifier, retrieves a packet descriptor for flow classification which conducts several operations. First, the fields of packet header (e.g., IP pairs and port pairs) are used to calculate a hash key indexing into a hash table in SRAM. Since the task requires much computing power, the hash unit is adopted to offload the computation. Second, if a hash hit occurred, the hash entry pointing to a *flow context* in SRAM is referred to enqueue a packet descriptor for inspection; otherwise the creation of both hash entry

and flow context is carried out at runtime. The flow context mainly consists of the SRAM address of flow queue keeping the packet descriptors, state of inspection and status flags. Then, the thread dispatcher thread chooses a packet descriptor among flow queues round-robinly and passes it to a packet inspector thread, performing the pattern matching. Once a packet payload is matched against a pattern, a message is delivered to the XScale through the XScale scratch ring for carrying out alert. Finally, the transmitter thread examines the transmitting scratch ring to determine whether a packet is waiting to be sent, fetching the packet's descriptor in SRAM and sends the entire packet in DRAM to TBUF for output.

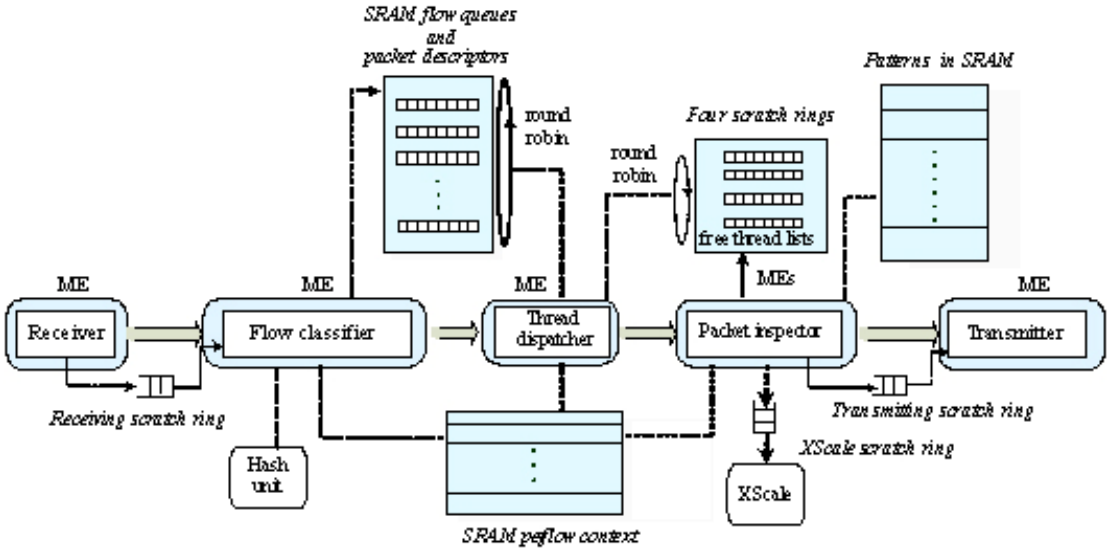

Fig. 3. The processing stages of an NIDS on IXP2400

In our implementation, a tentative allocation of MEs and threads is chosen based on the benchmarking of Snort. Each processing stage is allocated one ME except the packet inspector, which is given four MEs. That gives us totally four MEs and thirty-two threads to use for later adjustment and analysis. For thread allocation in the receiver, eight threads are evenly divided into four groups corresponding to four gigabit ports. Each port is served by two ordered threads to keep packets in order. As

for the transmitter, eight ordered threads are assigned to one gigabit port. For the flow classifier threads, classifying packets could take vastly different amounts of time when the hash collisions occur; considering the thread dispatcher threads, choosing among flow queues round-robinly needs to maintain the synchronization of a round-robin counter. Hence, we adopt eight ordered threads in both processing stages. In the packet inspector, unordered threads, rather than the ordered ones, are employed for the following reasons. First, the ordered thread may not be efficient in hiding the memory access latency due to each function in a processing stage can only be executing on one thread at any given time. Second, since a flow queue is served by one thread at a time, packets will never get out of order within that flow; meanwhile, ordered thread is adopted by processing stages before and after the packets being inspected. Hence the ordering of packets is maintained. Interaction between the thread dispatcher and packet inspector will be detailed in the section 4.4.2.

4.4 Algorithms Adopted and Packet Inspection

4.4.1 String Matching Algorithms

Packet inspection is a critical stage that influences the performance of an NIDS. Several string matching algorithms were proposed for improvement. However, coding microcode is difficult, since it depends heavily on hardware characteristics. Two popular algorithms, Aho-Corasick referred to as A-C and Wu-Manber referred to as W-M, are thus used because they are easy to implementation and adopted in most network applications, for example, Snort. The two algorithms typically consist of two phases: a pre-processing phase, which computes and builds necessary data structures in memory from input patterns, and an inspection phase, in which patterns are looked up against the packet payload. Nevertheless, the pre-processing phase is

time-consuming and typically done by the XSacle. For our implementation, we store the data structures in SRAM in order to reduce the memory access overhead. The operation of A-C involves state transitions of automaton. Therefore, we can record the state when finishing the inspection of a packet, and start from last state for the next one. Similarly, we keep the shift value instead of state for W-M so that patterns across multiple packets can be inspected.

4.4.2 Thread Dispatcher and Packet Inspector

Fig. 4 details the interactions between thread dispatcher and packet inspector. As mentioned before, the thread dispatcher thread chooses a packet descriptor among flow queues and passes it to the packet inspector thread, in which some operations are involved. First, two flags, namely *isEmpty* and *beingServed*, of a flow context are checked in each round. The former is to indicate if corresponding flow is occupied while the later is to denote whether that flow is being served by a thread. If it is occupied and not being served, a packet descriptor is assigned to a packet inspector thread; that is, a flow is served by only one packet inspector thread at a time, and thus preventing the state (for A-C) or shift value (for W-M) from being corrupted by another one. Further, the packet inspector thread examines a packet payload against thousands of patterns in SRAM, updating the state or shift value in the flow context. If a pattern is matched, a message is sent to the XScale, and then the packet is transmitted; otherwise the packet is sent to the transmitter directly. Finally, the packet inspector thread puts itself into a dedicated free thread list and waits for a signal from the thread dispatcher. The four free thread lists correspond to the four MEs and are implemented by four scratch rings. Notably, the packet inspector threads are dispatched round-robinly for the reason of load balance among MEs. Considering

timeout of a flow, a counter is associated with each flow to avoid the system resource being exhausted by excess flows, and is maintained by the XSacle. Once the counter is turned into zero, meaning that the lifetime of that flow is terminated; meanwhile, the flow queue as well as flow context and hash entry are eliminated.

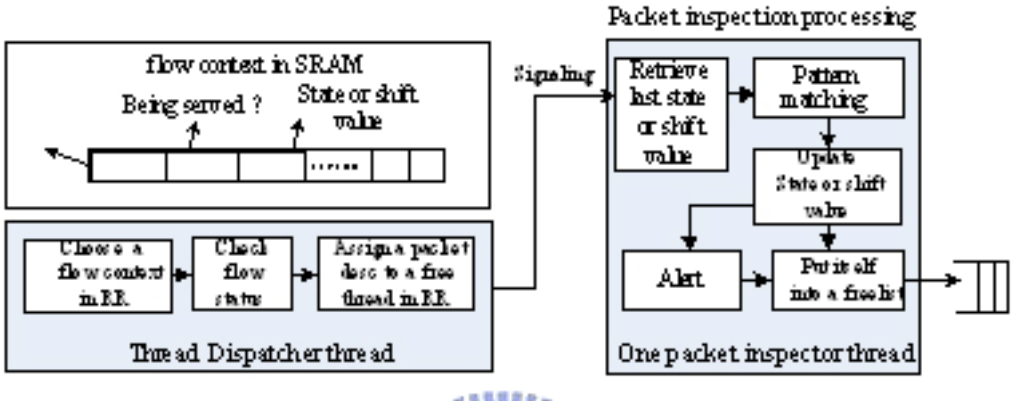

Fig. 4. Interaction between the thread dispatcher and packet inspector

Chapter 5. Performance Benchmark and Bottleneck Analysis

In this chapter, we evaluate the performance by externally and internally benchmarking the system implemented using two string matching algorithms. To have both MEs and memory, namely SRAM, well utilized, we investigate the appropriate numbers of *I* and *J* for the application. Since memory accesses account for a considerable portion in the packet processing, the feasibility of exploiting multiple memory banks for load balance is exploited.

5.1 Benchmark Setup

The XScale core in our design is responsible simply for the preprocessing and alerting; therefore, in this chapter we focus mainly on the performance of the MEs which are the main component that handles the most part of packet processing. Since the performance statistics including ME and memory utilizations can only be obtained by the simulator, we evaluate the performance figures through simulation. Consequently, the preprocessing phase originally done by the XScale is shifted to the receiver ME since the simulator doesn't comprise the XScale. Notably two MEs from two processing stages, the flow classifier and thread dispatcher, respectively, are borrowed in the analysis due to the dearth of MEs.

5.1.1 Patterns for Packet Inspection

Observing that 2475 patterns are used in the current Snort, we employ 2000 random patterns in which characters are generated uniformly according to the

guidelines discovered in [19]. The *shortest pattern length*, LSP, which is known as a major factor on the performance of string matching algorithms such as W-M, is set to four [20]. As mentioned previously, the pattern preprocessing phase is done directly by the receiver ME and the resulted data structure is stored in SRAM.

5.1.2 Simulator Setup

The IXP2400 Developer Workbench provides tools for compiling microC into microcode and a simulator called Transactor, for evaluating the performance. The simulator allows users to configure parameters. In our experiment, the clock of the ME is 600 MHz. The input interface of the MSF is divided into four gigabit ports, while the output interface is a four-gigabit one. The transmitter and receiver buffers are both 256 bytes. Four data streams of 64-byte TCP/IP packets with randomly generated payload are injected. The simulation lasts until 50000 packets are transmitted.

5.2 Effect of Improper ME/Thread Allocations

To investigate the effect of improper ME/thread allocations, we compare the performance, in terms of utilization, of A-C for different (*I*,*J*) combinations. As shown in Fig. 5, *I* and *J* can be configured while the total number of threads, $I \times J$, is fixed. Two outcomes are observed. First, the throughput is influenced mostly by $I \times J$, since the throughput remains unchanged for all (*I*,*J*) combinations. Second, the average ME utilization degrades while increasing *I*. This is because the $I \times J$, rather than the computing power, counts for the throughput, whereas the same traffic load is balanced by more MEs. The same explanation applies to the results for W-M in Fig. 6.

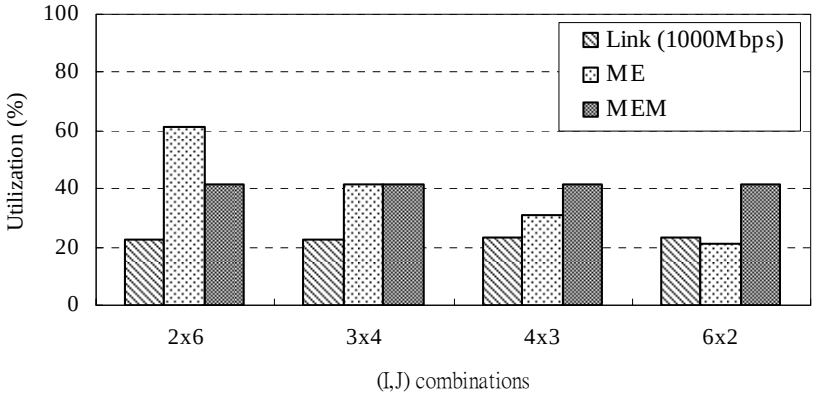

Fig. 5. The performance of A-C for different (*I*,*J*) combinations. The number of threads is fixed at 12.

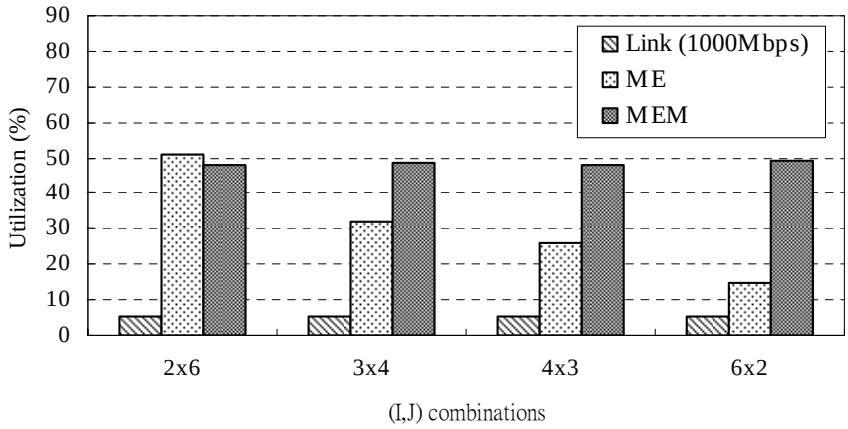

Fig. 6. The performance of W-M for different (*I*,*J*) combinations. The number of threads is fixed at 12.

Fig. 6 shows that the overall memory utilization of W-M is higher than the one of A-C while the average ME utilization is low. This is due to the high memory-access overhead of the former algorithm, as clarified in section 5.3.

5.3 Profiling on Memory Access and Computational Instructions

The average ME utilization and throughput of A-C are better than those of W-M. This is proven by the profiling of memory-access cycles required for handling a 64-byte packet, as shown in Fig. 7. Apparently W-M needs more memory-access cycles, referred to as *P*, than A-C as well as computational instruction cycles, referred to as *M*, as shown in Fig. 8. We estimate the ratio of $\frac{P}{M}$ for the number of patterns being 2000, $\frac{P}{M} \approx 0.06$ for A-C and $\frac{P}{M} \approx 0.02$ for W-M, respectively, meaning that the average ME utilization of W-M suffers from the heavy memory-access overhead except for the throughput.

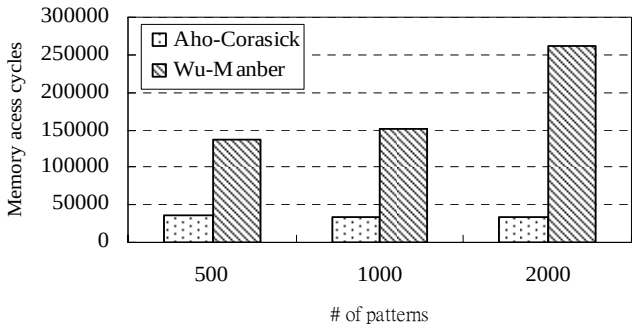

Fig. 7. Profiling of memory-access cycles for a 64-byte packet

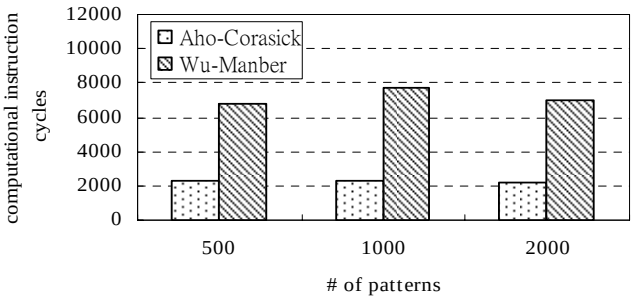

Fig. 8. Profiling of computational instruction cycles for a 64-byte packet

5.4 Estimating Optimal Numbers of MEs and Threads within Each ME

Fig. 9 depicts the performance of the two implementations by increasing number of MEs and therefore the total number of threads. Some observations can be made. First, the throughput of A-C is better due to less memory-access overhead. Second, for number of MEs being from one to four, the ME utilizations of both implementations are almost the same, implying that the number of threads per ME is insufficient. Third, initially, the throughputs of both implementations increase with a direct ratio to $I \times J$, namely number of threads. Nevertheless, the throughput increases slightly as I = 5 for W-M and I = 6 for A-C, respectively, because memory is almost fully utilized. Fourth, as I increases and memory utilization approaches 90%, the average ME utilization degrades, because the load making memory saturated is diluted by large I.

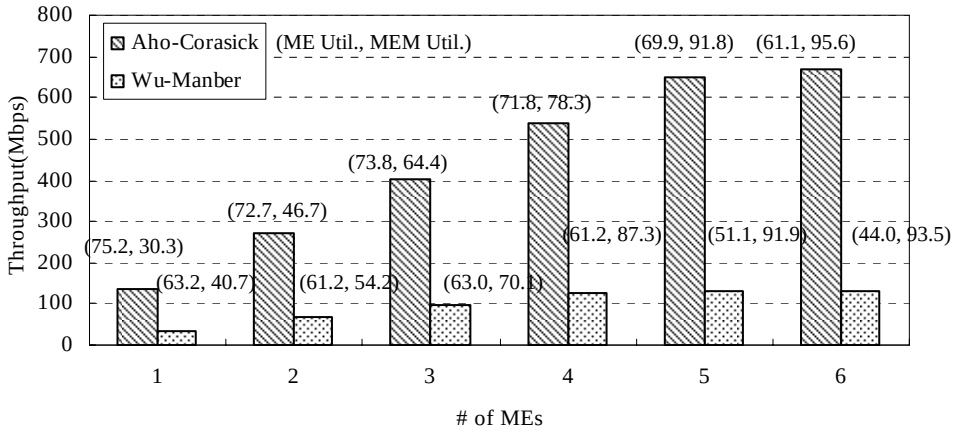

Fig. 9. The performance of A-C and W-M with different numbers of MEs (8 threads per ME)

We can also estimate a combination of (I,J) such that both ME and memory are best utilized. As we learn from Fig. 9, when memory utilization is above 90%,

increasing *I*, and therefore total number of threads contributes slightly to the performance and is not cost-effective. For example, the improvement of memory utilization from incorporating the sixth processor is about $95.6 - 91.8 \approx 3.8\%$ and $93.5 - 91.9 \approx 1.6\%$ for A-C and W-M, respectively. Hence, $5 \times 8 = 40$ threads should be enough cost-effectively for both algorithms to well utilize the memory. Nonetheless, the ME utilization is low when $I = 5$, meaning that the computing power is unnecessarily much and should be further reduced. We fix this problem by employing four MEs, rather than five, so that the average utilization of MEs shall become $\frac{69.9\% \times 5}{4} \cong 87.4\%$ (since $\frac{69.9\% \times 5}{3} \cong 116.5\% > 100\%$), and J can thus be estimated to $\frac{40}{4} = 10$. Similarly, a combination of $(3,13)$ can be derived for the W-M.

5.5 Bottleneck of SRAM Command Queues

The memory bottleneck can be further confirmed according to the history of SRAM command queues as shown in Fig. 10. From the figure, we can see that the queues are nearly full as $I = 4$. When $I = 5$, the queues start to overflow, indicating that no more requests can be served. Hence, threads issuing memory-access requests are blocked, resulting in the degradation of ME utilizations.

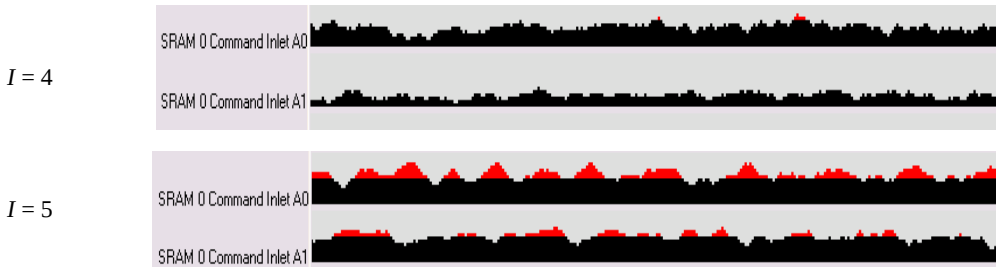

Fig. 10. History of SRAM command queues for W-M

5.6 Effectiveness of Multiple Memory Banks

One of the solutions to the memory bottleneck is to add more memory banks. To evaluate the benefit, we adopt two SRAM banks to store the data structures of the string matching algorithms. Table 1 shows that only minor improvement can be gained due to the difficulty of splitting the data structure, namely *goto* table, of A-C evenly into different memory banks. The W-M, on the contrary, benefits substantially (about 43.7%) from two banks as presented in Table 2. This is credited to the use of several tables which make the distribution of data a lot easier and more efficient to memory banks.

Table. 1. The performance of A-C with two memory banks when (*I*,*J*) = (6,8)

	One memory bank	Two memory banks
Avg. ME util. (%)	61.1	63.2
MEM util. (%)	95.6	95.2/1.8
Throughput (Mbps)	670.6	674.4

Table. 2. The performance of W-M with two memory banks when (*I*,*J*) = (6,8)

	One memory bank	Two memory banks
Avg. ME util. (%)	44.0	63.2
MEM util. (%)	93.5	70.0/57.2
Throughput (Mbps)	133.2	191.4

Chapter 6. Conclusion and Future Works

In this work, we elaborate the implementation of a memory-access intensive application, NIDS, over the IXP2400 network processor. We introduce the hardware platform, briefing the NIDS processing flow, and identify necessary processing stages to be mapped to the platform. NIDS includes a critical processing stage, the packet inspection, which are implemented with A-C and W-M. Some design issues including packet ordering and flow interleaving are solved, and thus patterns across multiple packets can be inspected appropriately. Finally we externally and internally benchmark the system aiming to observe the effect of the allocations of processors, threads, and memory banks, as well as possible bottlenecks.

The external and internal benchmark shows that the system can support up to 670 Mbps using the A-C and 133Mbps with the W-M. It is observed that given a certain application and algorithm, the throughput is influenced mostly by the total number of threads as long as the ME utilizations do not exceed 100%. Although enlarging $I \times J$ by adding more processors benefits the throughput, the ME utilization suffers. This is because the load saturating memory is diluted by the increased *I*, meaning that *J* instead should be extended. The bottleneck is then found to be the SRAM as the $I \times J$ expands and exceeds the upperbound, *k*, that *cost-effectively* utilizes the memory. With the upper-bound, we estimate an optimal (*I*, *J*), i.e. (4, 10) for the A-C and (3,13) for the W-M, respectively. In fact, supposed an application, algorithm and *k*, an optimal (*I*, *J*) can always be derived.

Two workarounds are suggested to solve the SRAM bottleneck, namely when $I \times J > k$. The first is to use multiple memory banks. Our result indicates that the performance gains a 43.7% improvement from two banks for W-M since the data structure itself makes it easy to be evenly distributed among banks. The other is to

adopt a multi-port memory which allows multiple simultaneous memory accesses. This is helpful especially to algorithms, such as the A-C, having data structures difficult to be uniformly split.

Two issues are to be investigated in the future. First, the influence, i.e. the real traffic rather than, synthetic one will be considered. Second, we tend to observe the impact of allocations of processors, threads, and memory banks on performance for computation-intensive applications.

References

[1] M. Roesh, Snort: The open source network intrusion detection system, http://www.snort.org.

[2] M. John and S. Smith, “Application-Specific Integrated Circuits,” Addison-Wesley Publishing Company, ISBN 0-201-50022-1, June 1997.

[3] P. C. Lekkas, “Network Processors: Architectures, Protocols and Platforms (Telecom Engineering),” McGraw-Hill Professional, ISBN 0071409866, July 2003.

[4] N. Shah and K. Keutzer, “Network Processors: Origin of species,” in Proceeding of ISCIS IVII, 2002.

[5] H. Bos and K. Huang, “A network instruction detection system on IXP1200 network processors with support for large rule sets,” Leiden Univeristry Techical Report 2004-02.

[6] Intel® IXP12XX Product Line of Network Processors, http://www.intel.com/design/network/products/npfamily/ixp1200.htm.

[7] A. Aho and M. Corasick, “Efficient string matching: An aid to bibliographic search, “ Communications of the ACM, Vol. 18, Issue 6, P.333-340, 1975.

[8] C. Clark, et al., “A Hardware Platform for Network Intrusion Detection and Prevention," In Proceedings of the 3rd Workshop on Network Processors and Applications (NP3), Madrid, Spain, February 2004.

[9] IXP2400 Data Sheet, Intel document number 301164-011, February 2004.

[10] Matthew, et al., “The Next Generation of Intel IXP Network Processors," Intel Technology Journal Vol.6 Issue 3, 2002.

[11] S. Wu and U. Manber, “A fast algorithm for multi-pattern searching. Technical Report TR94-17, “ Department of Computer Science, University of Arizona.

[12] Intel XScale Microarchitecture, http://www.intel.com/design/intelXScale.

[13] POS PHY Level 3 Link Reference Design, http://www.latticesemi.com/products/devtools/ip/refdesigns/pos_phy.cfm.

[14] CSIX-L1: Common Witch Interface Specification, http://www.npforum.org/csixL1.pdf.

[15] S. Lakshmanamurthy, et al., “Network Processor Performance Analysis Methodology“ Intel Technology Journal Vol.6 Issue 3, 2002.

[16] M. Fisk and G. Varghese, “Applying Fast String Matching to Intrusion Detection,” SEP 2002.

[17] U. Naik, et al., “IXA Portability Framework: Preserving Software Investment in Network Processor Applications, “ Intel Technology Journal Vol.6 Issue 3, 2002.

[18] E. J. Johnson and A. R. Kunze, “IXP2400/2800 Programming– The Complete Microengine Coding Guide,” Intel Press, April 2003.

[19] S. Antonatos, K. G. Anagnostakis, M. Polychronakis, and E. P. Markatos, “Performance Analysis of Content Matching Intrusion Detection Systems,” Proceedings of the International Symposium on Applications and the Internet (SAINT2004), January 2004.

[20] Rong-Tai Liu, Nen-Fu Huang, Chih-Hao Chen and Chia-Nan Kao, “a fast string-matching algorithm for network processor-based intrusion detection system,“ ACM Transactions on Embedded Computing Systems, Vol 3, Issue 3, P.614-633, August 2004.

[21] Ying-Dar Lin, Yi-Neng Lin, Shun-Chin Yang and Yu-Sheng Lin, "DiffServ Edge Routers over Network Processors: Implementation and Evaluation," IEEE Network, Special Issue on Network Processors, July 2003.