Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1030–1036
The Journal of Systems and Software
www.elsevier.com/locate/jss

Resource allocation in network processors for network intrusion prevention systems

Yi-Neng Lin [a,*], Yao-Chung Chang [a], Ying-Dar Lin [a], Yuan-Chen Lai [b]

[a] Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
[b] Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan

Received 29 May 2006; received in revised form 14 December 2006; accepted 3 January 2007
Available online 6 February 2007

Abstract

Networking applications with high memory access overhead gradually exploit network processors that feature multiple hardware multithreaded processor cores along with a versatile memory hierarchy. Given rich hardware resources, however, the performance depends on whether those resources are properly allocated. In this work, we develop an NIPS (Network Intrusion Prevention System) edge gateway over the Intel IXP2400 by characterizing/mapping the processing stages onto hardware components. The impact and strategy of resource allocation are also investigated through internal and external benchmarks. Important conclusions include: (1) the system throughput is influenced mostly by the total number of threads, namely $I \times J$, where I and J represent the numbers of processors and threads per processor, respectively, as long as the processors are not fully utilized, (2) given an application, algorithm and hardware specification, an appropriate (I, J) for packet inspection can be derived and (3) the effectiveness of multiple memory banks for tackling the SRAM bottleneck is affected considerably by the algorithms adopted.

Keywords: Network intrusion and detection system; Network processor; Resource allocation; Benchmark; Bottleneck

1. Introduction

Networking applications offering security and content-aware processing demand powerful hardware platforms to achieve a high performance. General-purpose processors are often adopted for memory-access-intensive applications such as network intrusion detection systems (NIDS) (Roesh, 2006), which accumulate data and analyze traffic to identify possible security breaches. However, such processors have a high cost and poor throughput owing to that the processor utilization is low, because of the heavy memory access overhead. Conversely, the application-specific integrated circuits (ASICs) (John and Smith, 1997) can satisfy the performance requirement with a circuitry designed for strict guarantees on memory access latency using pipelined architecture and embedded memory. Nevertheless, the lack of re-programmability reduces the appeal of ASICs.

Network processors (Lekkas, 2003; Lin et al., 2003) are emerging as an alternative means of resolving the above problems for their multithreaded multiprocessor architecture and the flexibility. Multiple processors allow simultaneous data-plane processing of multiple packets on a cluster of processors. Moreover, the hardware threads with a miniaturized context switch overhead can conceal the memory access latency (Shah and Keutzer, 2002) and therefore increase the throughput. The re-programmability makes functional adaptations much easier in a network processor than in an ASIC, which must need to be re-designed.

Despite rich hardware resources of network processors such as processors, threads and memories, the allocation of components that further influence their utilizations needs to be carefully planned. For memory-access intensive-applications, Bos and Huang (Bos and Huang, 2004)

* Corresponding author. Tel.: +886 988 211430.
E-mail address: ynlin@cs.nctu.edu.tw (Y.-N. Lin).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.01.032

have implemented an NIPS on an Intel IXP1200. The prototype comprises only the receiver and packet processing using the Aho–Corasick (Aho and Corasick, 1975) algorithm, but does not support multiple flows or inspections of pattern stretching for more than two packets. Clark (2004) designed a network intrusion detection prevention system (NIPS) that incorporates an IXP1200 for header processing and an field programmable gate array (FPGA) as the signature-matching engine. The bottleneck is the bus connecting them. Kang et al. (2006) further exploited the high-end IXP2800 and TCAMs to achieve a multi-gigabit NIPS for a core network. Nevertheless, these studies do not thoroughly consider the resource allocation strategies and, therefore, could lead to low component utilization.

This work implements an NIPS edge gateway, which is more frequently adopted than the core device, over an Intel IXP2400 (Intel, 2004) processor, which has a similar architecture to most network processors. The system detects, rather than prevents, the attacks by sniffing flows through an IP/port sockets, and writes the results into a log file. The system includes two signature-matching algorithms, Aho–Corasick and Wu–Manber (Wu and Manber, 1994), due to their popularity in many security-related implementations, such as Snort. Several software components, called the *processing stages* (Adiletta, 2002) are characterized, in which a tentative processor/thread allocation is applied. After implementation, both external and internal benchmarks are then performed to address the issues considered below:

- *Effect of improper ME/Thread allocations on system performance*: Two factors affect the performance of an application, i.e., the computing power and the memory access latency. The first is determined by the number of processors in use, given by I, while the impact of the second can be alleviated by adjusting the total number of threads in use, given by $I \times J$ (Lakshmanamurthy, 2002), where J denotes the number of threads per processor. Since the total number of processors is fixed in the hardware platform, the effect of an allocation (I, J), especially an improper one, on the system performance is of priority concern.
- *Derivation of an appropriate I and J*: An appropriate (I, J) combination should be calculated, given a certain application and hardware spec such as clock rate and memory service rate, and regardless of the limit on the number of MEs and number of threads per ME in the platform.
- *Effectiveness of employing multiple memory banks*: Multiple memory banks reduce the average memory access latency by alleviating the queuing effect. Nonetheless, the effectiveness of doing so is not clarified.

This article is organized as follows. Section 2 describes the hardware architectures of IXP2400. Section 3 briefly describes the two matching algorithms, and elaborates the design and implementation of the proposed system. Section 4 presents the results and observations from external and internal benchmarks. Conclusions are drawn in Section 5.

2. Hardware architecture of IXP2400

As shown in Fig. 1, the IXP2400 comprises several components, categorized as follows.

2.1. Multithreaded multiprocessor architecture

The IXP2400 features nine programmable processors, which are one Intel XScale core and eight microengines (MEs), operating at 600 MHz. The Intel XScale core is responsible for housekeeping functions such as table initialization and exception handling for control-plane packets, which contain control messages such as "ICMP unreachable". Data-plane processing, which performs the actual manipulation such as checksum calculation, encryption/decryption and forwarding, and which accounts for the largest part in packet processing, is implemented on MEs. Every ME has eight hardware threads, each with its own register set and program counter to support fast context switching when memory accesses occur.

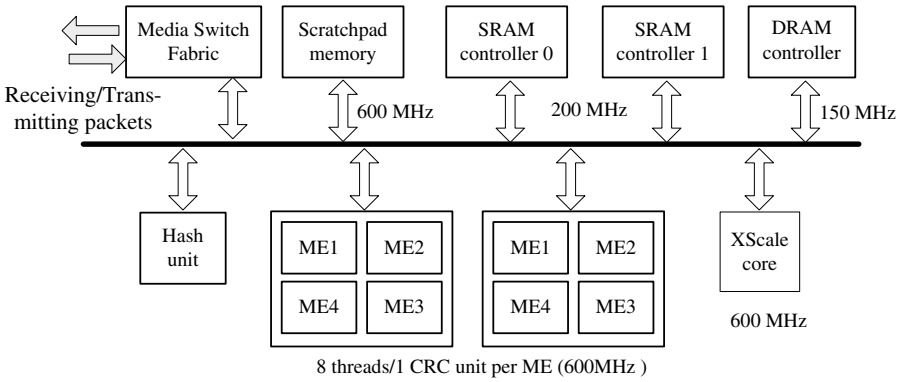

Fig. 1. Hardware architecture of IXP2400.

2.2. Hierarchical memory structure

To ease the memory access overhead, IXP2400 exploits four memories, DRAM, SRAM, scratchpad, and local memory in an ME, given tradeoffs between size and latency. IXP2400 has one channel of Double Data Rate (DDR) memory running at 150 MHz. The channel can support up to 2 GB of DRAM, yielding enough capacity for storing packets received by MEs directly from receive buffer (RBUF) at the Media Fabric Switch (MSF). Two channels of Quad Data Rate (QDR) SRAM running at 200 MHz are also provided, and each channel can hold up to 16 MB of data. The SRAM primary accommodates packet descriptors for locating packets in DRAM, queue descriptors and other frequently employed data structures, such as routing tables and matching signatures. The on-chip 16 KB scratchpad memory, i.e., a memory array with the decoding and the column circuitry logic, consumes less power than ordinary cache memories, operates in the form of rings and provides a similar capability to SRAM. Meanwhile, the 2560-word local memory functions as the data cache.

2.3. Detailed packet flow in IXP2400

Fig. 1 illustrates the processing flow of an ordinary packet. The MSF of an IXP2400 partitions an arriving packet into several smaller chunks called mpackets, which can be configured to 64, 128, and 256 bytes in size for easy manipulation and, then, places them into the RBUF. The threads of the MEs dedicated for receiving subsequently perform the reassembly of mpackets, and move the resulting packets directly from the RBUF into DRAM, in which the MEs and XScale core carry out further operations. The exceptions and housekeeping are handled by the XScale core through the interrupt and message queue mechanisms during processing at the MEs. Finally, the transmission process is simply 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 transmit buffer (TBUF).

3. Design and implementation

3.1. Adopted algorithms

Packet inspection, i.e., the detection phase, is a critical stage that affects the performance of an NIPS. This study employs two conventionally adopted algorithms, Aho–Corasick (A–C) and Wu–Manber (W–M), owing to their popularity in many applications such as Snort, and are easily implemented. The A–C exploits a pre-computed finite automation stored as the *goto table*, and accepts all the strings in the pattern set. Each character is then sequentially fed to the automation, which tracks partially matched patterns through state transition. The W–M reads a block of characters, rather than one character at a time, and looks up the pre-calculated *hash* and *shift* tables to calculate the shift distance. The signatures transformed into an automation or shift table are stored in SRAM for fast retrieval. To support statefull inspection, the final state is recorded immediately after a packet is processed for later scrutiny of the succeeding packet in the same flow. Similarly, the shift distance for W–M, is kept so that patterns across multiple packets can be inspected.

3.2. Mapping processing stages to the hardware platform

Fig. 2a shows the processing stages, namely the receiver, flow classifier, thread dispatcher, packet inspector and transmitter, of an NIPS, as well as the task and resource allocation for IXP2400. Upon receiving a packet from an input port, the payload is moved from RBUF to DRAM. Additionally, the corresponding packet descriptor is stored in SRAM. Moreover, a duplicate descriptor is passed onto the next stage via the receiving scratch ring.

The flow classifier subsequently retrieves a packet descriptor for flow classification. The descriptor operates as follows. First, the IP and port pairs in the packet are used to compute a hash key for indexing in the hash table in SRAM to verify whether the flow to which the packet belongs exists. Since this task requires a high computational power, a hash unit is adopted to offload the overhead. If a hash hit occurs, then the hash entry pointing to a flow context in SRAM is referred to enqueue the packet descriptor for inspection. Otherwise, an entry for the new flow is created in the hash table. The dispatcher thread then chooses a flow queue and, then, dispatches a free inspector thread to handle the first packet in the queue. A message is delivered to the XScale through the XScale scratch ring to signal an alert when a packet payload is matched against a pattern. Otherwise, the transmitter thread examines the transmitting scratch ring to determine whether an inspected packet is waiting to be sent. If yes, then the thread fetches the packet descriptor in SRAM, and transmits the entire packet in DRAM to TBUF for output.

The proposed implementation tentatively allocates MEs and threads based on the processing stages and the benchmark results of Snort, which argue that at least 31% of the total processing time is consumed by the inspection phase (Fisk and Varghese, 2001). Therefore, each processing stage is allocated one ME except the packet inspector, which is given four MEs. Hence, 32 threads are available for later adjustment. For thread allocation in the receiver, eight threads are evenly divided into four groups corresponding to four gigabit ports. Every port is served by two *ordered threads* (Johnson and Kunze, 2003), a mechanism requiring threads to execute the functions of a processing stage in order to keep the packet ordering. A thread is signaled only after its predecessor finishes those functions. In the transmitter, eight ordered threads are assigned to one four-gigabit port. This study employs eight ordered threads in both the classifier and dispatcher stages

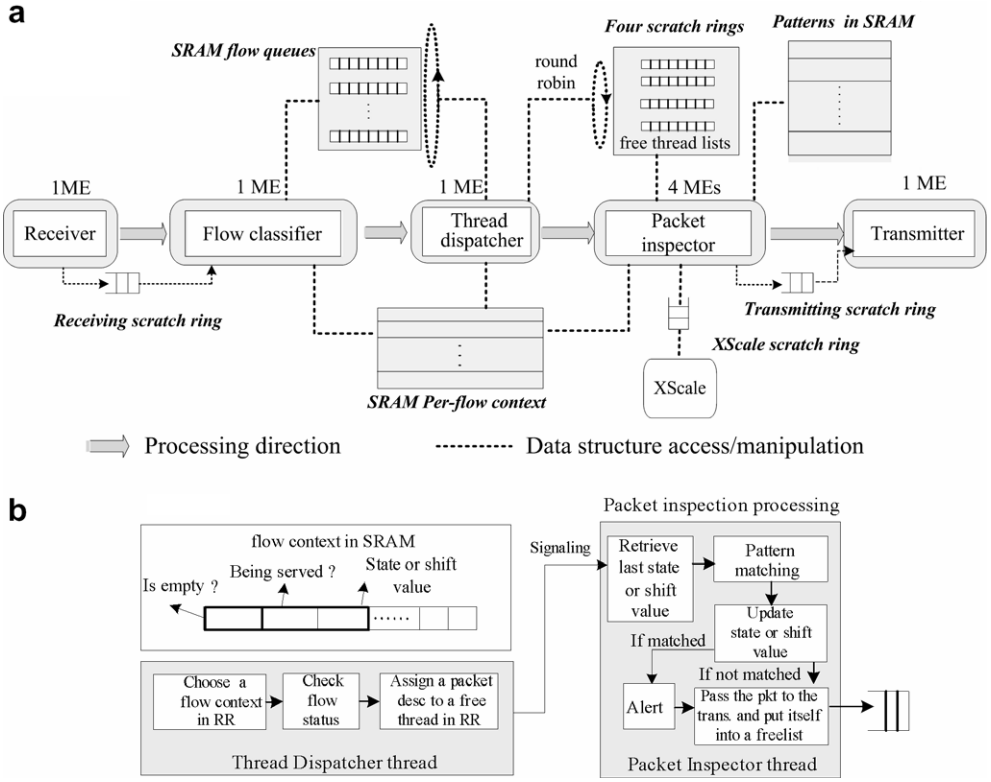

Fig. 2. The (a) processing stages and (b) interaction between the thread dispatcher and packet inspector. Notably all eight threads are active in an ME, though only one of them is actually processing packets.

for the following two reasons: (1) classifying packets could take significantly different amounts of time due to hash collisions, and could therefore lead to out-of-order packets, and (2) serving flow queues with multiple dispatcher threads requires mutual exclusion on the counter, which records the index of the queue being served.

3.3. Thread dispatcher and packet inspector

Fig. 2b illustrates the detailed interaction between the thread dispatcher and the packet inspector. As mentioned in Section 4.2, a flow queue is selected, and the first packet descriptor in that flow is passed to an inspector thread chosen from the free thread lists of the four MEs. This process involves some operations. First, two flags, *isEmpty* and *beingServed*, of a flow context are checked in each round. The first flag indicates whether the corresponding flow is empty, while the second indicates whether that flow is being served by a thread. If the flow is not empty ($isEmpty = 0$) and not being served ($beingServed = 0$), then a packet descriptor is assigned to an inspector thread, and the *beingServed* flag is consequently set to 1. This ensures that a flow is served by only one inspector thread at a time, thus preventing the state (for A–C) or shift distance (for W–M) of the flow from being altered by other threads. The inspector thread then examines a packet payload against the patterns in SRAM, and updates the state or shift distance in the flow context accordingly. If no pattern is matched, the packet is passed to the transmitter thread to be sent out; otherwise the XScale is notified of a match. Finally, the packet inspector thread places itself into the free thread list, waiting for the next signal from the dispatcher. The four free thread lists implemented by the four scratch rings correspond to the four MEs. The inspector threads are dispatched from the MEs for load balancing. To prevent the system resource from being exhausted by excess idle flows, a timeout counter maintained by the XScale is associated with each flow. The flow queue as well as the flow context and hash entry is removed once the counter times up.

Further support for TCP stream reassembly is yet to be implemented in the receiver stage. Local memory could be exploited to store the sequence numbers of out-of-order packets temporarily held in the SDRAM. The ME compares the sequence number of a newly arriving data packet against the local memory entries to determine whether it plugs any of the sequence holes. If all holes are filled up, then those packets are enqueued to the corresponding flow queue in SRAM.

4. Performance benchmark and bottleneck analysis

This section assesses the performance of the system by externally and internally benchmarking the system implemented using two string-matching algorithms. The appropriate (I, J) for the critical packet inspection stage was explored to ensure that both MEs and SRAM memory were well utilized. Additionally, the feasibility of exploiting

multiple memory banks for load balance is discussed since the memory access overhead accounts for a considerable portion of the packet processing.

4.1. Benchmark setup

The clock of the ME in our experiment was 600 MHz. The input interface of the MSF was divided into four gigabit ports, while the output interface was a four-gigabit port. Four data streams of 64-byte TCP/IP packets with randomly generated payload were injected. All simulations lasted for 50 000 packets.

With 2475 patterns used in the current Snort, 2000 random patterns were applied, with characters generated uniformly according to the guidelines discovered in (Antonatos et al., 2004). The length of shortest pattern (LSP), which is known to be a major factor on the performance of string matching algorithms such as W–M, was set to 4 characters (Liu et al., 2004).

4.2. Effect of improper ME/thread allocations on system performance

Improper ME/thread allocations can yield a poor performance. To investigate the effect of such allocations on the system, the performances of A–C and W–M were compared, in terms of utilization, for different (I, J) combinations. As revealed in Fig. 3, I and J can be configured, while the total number of threads, $I \times J$, is fixed at 12. Some observations are made. First, the throughput, i.e., the link utilization, was influenced mostly by $I \times J$, rather than I, since it remains unchanged for different (I, J) combinations. This finding suggests that additional threads are needed to advance the memory utilization and the throughput. Second, the average ME utilization degraded with rising I, because the same traffic load was balanced by extra MEs. Third, the throughput of W–M was only one-fourth of that of A–C owing to the relatively high processing overhead of W–M, as illustrated in Fig. 4.

Fig. 4 profiles the total computational cycles, denoted by P, and the memory access cycles, represented by M, required by A–C and W–M to handle a 64-byte packet. The figure indicates that the sum of P and M in W–M is approximately 4 times of that in A–C. This explains the relatively low throughput of W–M. Moreover, the memory access overhead dominates the processing time of a packet, namely 94% ($\frac{34920}{2309+34920} \cong 94\%$ when # of patterns = 500) for A–C and 98% ($\frac{137340}{6763+137340} \cong 98\%$ likewise) for W–M. Fortunately, this imbalance is handled through multithreading, which narrows the difference in utilization of MEs and memory.

4.3. Estimating an appropriate (I, J) pair through bottleneck analysis

Fig. 5 depicts the performance of the two implementations by increasing the number of MEs and thus the total number of threads. Some observations can be made. First, the system can scale up to 670 Mbps when implemented using the A–C and 133 Mbps using the W–M. The throughput of A–C is better due to the low computational and memory access overhead. Second, the ME utilizations

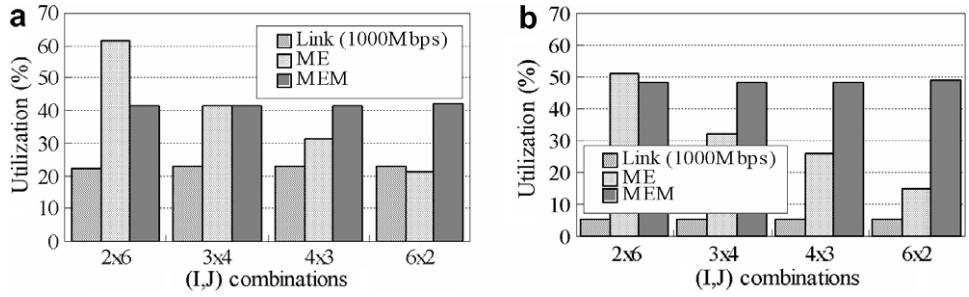

Fig. 3. Performance of: (a) A–C and (b) W–M for different (I, J) combinations. Total number of threads is fixed at 12.

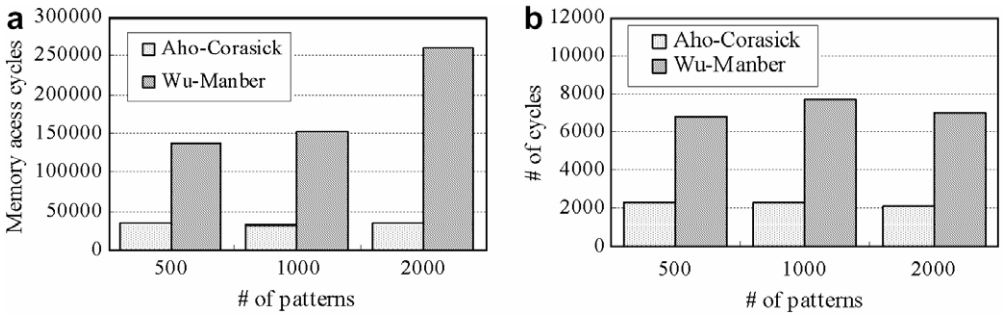

Fig. 4. Profiling of: (a) total memory access cycles and (b) total computational cycles for processing a 64-byte packet.

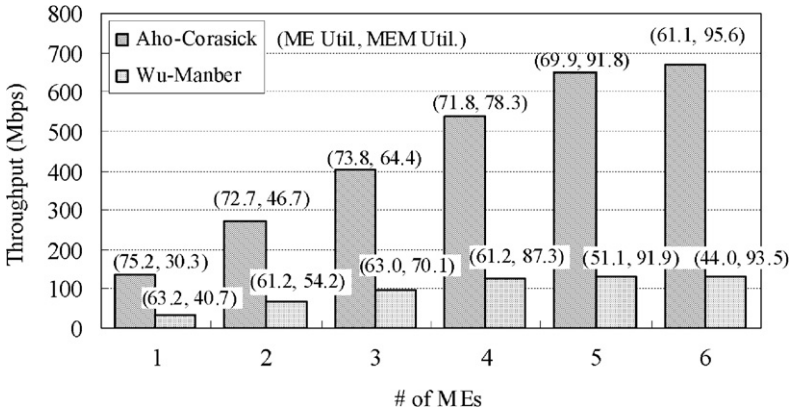

Fig. 5. The performance of A–C and W–M with different numbers of MEs (eight threads per ME).

of both implementations are far from fully utilized with 1–4 MEs, implying that the number of threads per ME is too small to utilize the MEs effectively. Third, the throughputs in both implementations will increase in direct proportion to $I \times J$. Nevertheless, the throughput rises slightly as I increases to 6 for A–C and to 5 for W–M, because the memory is almost fully utilized, and is therefore the bottleneck. Fourth, the average ME utilization degrades as I increments and the memory utilization approaches 90%, since the load that stops increasing due to the memory bottleneck is diluted by the rising I.

A combination of (I, J) that enables both MEs and memory to be effectively utilized can be further estimated through the bottleneck analysis. Fig. 5 demonstrates that increasing I, and therefore increasing the total number of threads slightly improves the performance when memory utilization is above 90%. For instance, the improvement of memory utilization from incorporating the sixth processor is merely $95.6 - 91.8 \approx 3.8\%$ and $93.5 - 91.9 \approx 1.6\%$ for A–C and W–M, respectively. Hence, the appropriate upperbound, k, $5 \times 8 = 40$ threads should be in both algorithms if the memory is to be utilized effectively. Nonetheless, the ME utilization is low when $I = 5$, meaning that the computing power is unnecessarily high and should be further reduced. This problem is fixed by employing four MEs, rather than five, so that the average utilization of MEs becomes $\frac{69.9\% \times 5}{4} \cong 87.4\%$ (since $\frac{69.9\% \times 5}{3} \cong 116.5\% > 100\%$), and J is thus estimated as $\frac{40}{4} = 10$. Similarly, a combination of (3, 13) can be derived for W–M. The derivation can thus be generalized as follows:

APPR-$I\_J(n = 1,J')//n$: number of processors used, J': threads per processor in the platform

```
while(true)
  if m_util(n,J')! ≅ 100%
    n++;
  elsif m_util(n,J') ≅ 100% and p_util(n,J') ≅ 100%
    return(n,J');
  else //m_util(n,J') ≅ 100% and p_util(n,J')! ≅ 100%;
    k = n × J';
```

Table 1
Performance of (a) A–C and (b) W–M with one and two memory banks, respectively. (I, J) = (6, 8)

	One memory bank	Two memory banks
(a)		
ME util. (%)	61.1	63.2
MEM util. (%)	95.6	95.2, 1.8
Throughput (Mbps)	670.6	674.4
(b)		
ME util. (%)	44.0	63.2
MEM util. (%)	93.5	70.0, 57.2
Throughput (Mbps)	133.2	191.4

return$\left(i', \frac{k}{i'}\right)$, where $i' < n$ and $\frac{p\_util(n,J') \times n}{i'}$ is *smaller and closest to* 100%.

Empirically, $x\_util(n,J') \cong 100\%$ means that $x\_util(n,J') > 90\%$.

4.4. Effectiveness of multiple memory banks

One solution to the memory bottleneck is to add more memory banks. To evaluate the benefit of this approach, the signatures were stored in two SRAM banks. Table 1a indicates that very limited improvement can be gained for A–C due to the difficulty of splitting the *goto* table evenly into different memory banks. Conversely, W–M benefits substantially (by about 43.7%) from having two banks as presented in Table 1b, because of the use of two tables that simplifies the distribution of data.

5. Conclusions and future work

This study investigates the resource allocation methodologies of network processors by implementing and evaluating a memory access intensive application, the network intrusion prevention system. The hardware platform, IXP2400 is introduced, and the necessary software processing stages to be mapped to the platform are identified. Among these processing stages, the packet inspection is implemented with

the Aho–Corasick and Wu–Manber algorithms. External and internal benchmarks are then undertaken to examine the effect of resource allocation and the possible bottlenecks. The observations should be applicable to other network processors because of similar components and architectures.

Benchmark results show that the system can scale up to 670 Mbps using Aho–Corasick and 133 Mbps using Wu–Manber. The throughput is influenced mostly by the total number of threads while the ME utilizations are not fully utilized. SRAM is then found to be the bottleneck, since $I \times J$ exceeds the upper bound k of appropriate memory usage. The upper bound can be further adopted to estimate an appropriate (I, J) combination for both algorithms. An appropriate (I, J) can always be derived when given an application, algorithm and hardware specifications, such as the memory service time and processor clock rate.

Two work arounds are proposed to solve the SRAM bottleneck. The first is to employ multiple memory banks, which yields a 43.7% improvement in Wu–Manber, since the signature can easy be evenly distributed among the banks. The other is to apply a multi-port memory that enables simultaneous memory accesses. The proposed approach appears to be particularly helpful to algorithms, such as Aho–Corasick, that have data structures that are difficult to split uniformly among banks.

Two issues will be investigated in the future. First, although synthesized traffic is acceptable given the level of error (Antonatos et al., 2004), real traces are preferable. Another issue is to consider the allocation strategy for computational-intensive applications.

References

Adiletta, M. et al., 2002. The Next Generation of Intel IXP Network Processors. Intel Technology Journal 6 (3).

Aho, A., Corasick, M., 1975. Efficient string matching: an aid to bibliographic search. Communications of the ACM 18 (6), 333–340.

Antonatos, S., Anagnostakis, K.G., Polychronakis, M., Markatos, E.P., 2004. Performance analysis of content matching intrusion detection systems. In: Proceedings of the International Symposium on Applications and the Internet (SAINT2004).

Bos, H., Huang, K., 2004. A Network Instruction Detection System on IXP1200 Network Processors with Support for Large Rule Sets. Leiden Univeristry Techical Report.

Clark, C., et al., 2004. A hardware platform for network intrusion detection and prevention. In: Proceedings of the 3rd Workshop on Network Processors and Applications (NP3).

Fisk, M., Varghese, G., 2001. Applying fast string matching to intrusion detection. Technical Report CS2001-0670, UCSD.

Intel, 2004. IXP2400 Data Sheet. Intel document number 301164-011.

John, M., Smith, S., 1997. Application-Specific Integrated Circuits. Addison-Wesley Publishing Company, ISBN 0-201-50022-1.

Johnson, E.J., Kunze, A.R., 2003. IXP2400/2800 Programming – The Complete Microengine Coding Guide. Intel Press.

Kang, S.-M., Song, I.-S., Lee, Y., Kwon, T.-G., 2006. Design and implementation of a multi-gigabit intrusion and virus/worm detection system. In: Proceedings of the International Conference on Communications (ICC'06).

Lakshmanamurthy, S. et al., 2002. Network processor performance analysis methodology. Intel Technology Journal 6 (3).

Lekkas, P.C., 2003. Network Processors: Architectures, Protocols and Platforms (Telecom Engineering). McGraw-Hill Professional Publishing Co.

Lin, Y.-D., Lin, Y.-N., Yang, S.-C., Lin, Y.-S., 2003. DiffServ edge routers over network processors: implementation and evaluation. IEEE Network 17 (4), 28–34.

Liu, R.-T., Huang, N.-F., Chen, C.-H., Kao, C.-N., 2004. A fast string-matching algorithm for network processor-based intrusion detection system. ACM Transactions on Embedded Computing Systems 3 (3), 614–633.

Roesh, M., 2006. Snort: the open source network intrusion detection system. Available at <www.snort.org>.

Shah, N., Keutzer, K., 2002. Network processors: origin of species. In: Proceedings of the Seventeenth International Symposium on Computer and Information Sciences ISCIS IVII.

Wu, S., Manber, U., 1994. A fast algorithm for multi-pattern searching. Technical Report TR94-17, Department of Computer Science, University of Arizona.