Thread Allocation in Chip Multiprocessor Based Multithreaded Network Processors

Yi-Neng Lin, Ying-Dar Lin
Department of Computer Science, National Chiao-Tung University, Hsinchu, Taiwan
{ynlin,ydlin}@cs.nctu.edu.tw

Yuan-Chen Lai
Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan
laiyc@cs.ntust.edu.tw

Abstract*—This work tries to derive ideas for thread allocation in Chip Multiprocessor (CMP)-based network processors performing general applications by Continuous-Time Markov Chain modeling and Petri net simulations. The concept of P-M ratio, where P and M indicate the computational and memory access overhead when processing a packet, is introduced and the relation to thread allocation is explored. Results indicate that the demand of threads in a processor diminishes rapidly as P-M ratio increases to 0.066, and decreases slowly afterwards. Observations from a certain P-M ratio can be applied to various software-hardware combinations having the same ratio.*

1. Introduction

The advantages of traditional multithreaded-multiprocessor architectures are three-fold: increasing the computing power considerably by interconnecting a number of processing elements; sharing limited memory resource with others and thus form a distributed shared-memory, and tolerating the memory access overhead by multithreading. However, the memory subsystem tends to be the performance bottleneck because of the burden of long access delay. Fortunately, today's technology has made it possible to put several processors and memory banks on a single chip such that memory access latency is significantly reduced. This kind of architectures is emerging as *chip multiprocessor (CMP) based multithreaded processors* [1-3].

Though the architecture is promising in its scalability and extensibility, especially in the form of some network processors, the determination of architectural parameters such as numbers of processors, threads in a processor, and memory banks, is not trivial given a specific application and a performance target. Furthermore, since one proper configuration today may not be suitable tomorrow due to different evolving speeds of manufacturing technologies of the functional units, some general guidelines may be demanded for efficient and appropriate parameter determination.

A number of recent works concerning the modeling of CMP based multithreaded network processors can be found in [6-10]. Though detailed parameters are included and programming paradigms are analyzed, the discussion of thread allocation is substantially ignored. Lakshmanamurthy et al. propose a methodology for analyzing the performance of the Intel IXP2400 [18]. But they focus only on the validation of the system performance; the processor and memory utilizations are not addressed and design guidelines are not comprehensively investigated.

In this work, we aim to unveil possible hints for future design of the architecture by (1) developing a preliminary analytical model and (2) building a Petri net simulation environment for model validation and design implications observation. Our approach considers both memory and ready queuing effects that are often ignored in other works. Though the validated analytical model is found not scalable enough for deep observations, the simulation results demonstrate interesting design implications. We propose a concept named *P-M ratio*, where *P* and *M* represent the computational and memory access overheads of an application, and estimate a projection between P-M ratios and the corresponding appropriate number of threads in a processor. Workarounds to the memory bottleneck occurring at small P-M ratios are also discussed.

The rest of this article is organized as follows. Section 2 reviews related works and introduces the concept of thread allocation schemes. Section 3 elaborates the analytical model. Section 4 details the construction of the Petri net simulation environment, validates the analytical model, and presents some interesting simulation results. Conclusive remarks and

This work was supported in part by the Taiwan Natioanl Science Council's Program of Excellence in Research, and in part by grants from Cisco and Intel

future work are given in section 5.

2. Architectural Assumption on the Thread Allocation Scheme

Thread allocations should be carefully discussed before analyzing the architecture. Four thread allocation schemes are possible to real implementations, in which at most one thread is active in a processor. The first is that a thread is responsible for a complete packet processing. Nonetheless, this scheme may require intricate inter-thread communications in order to maintain the packet ordering in a flow.

Figure 1 presents another two schemes, the homogeneous and heterogeneous thread allocations. In the homogeneous allocation, all threads in a processor belong to the same type, e.g. receiver, scheduler, transmitter, etc. Each thread in a processor deals with only part of the packet processing and after that, it signals a certain thread in the succeeding processor for further processing. A thread in a processor may have either fixed or dynamic task assignment, namely it may stick to a certain input port or may support other ports whenever necessary. Notably, since all threads in a processor are of the same type, this scheme has a relaxed requirement for instruction memory size and exhibits desirable cache locality. Nonetheless, the processing load is unlikely to be distributed to processors evenly, and packet ordering is hard to maintain.

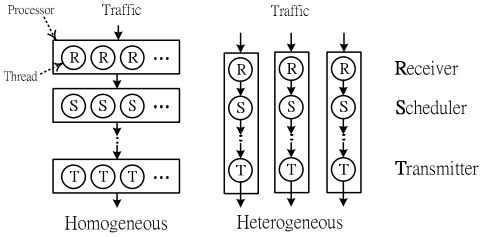


Fig. 1. Homogeneous and heterogeneous thread allocations. At most one thread is active per processor.

This situation can be avoided with the heterogeneous allocation, where traffic is assigned to processors by some load-balancing hardware [11] and mechanisms. In this scheme, threads in a processor belong to different types and are supposed to take an equal charge in the packet processing. Though a large instruction memory is needed to support various tasks, it will not be a problem because general header processing applications consist of less than 5K [12]instructions, which has already been supported in many commercial products such as the Intel IXP2400 [13]. Another edge of the scheme is the minor synchronization overhead, since the inter-thread communication is done using global registers in the processor. For the reasons discussed above, we take the heterogeneous allocation as the base assumption in our model throughout this work.

3. Analytical Model

In this section we present an approximate analysis of the architecture using the Continuous-Time Markov chain. We define the state space of the model, derive the transition rates, and solve the model. In addition to the heterogeneous allocation determined in the previous section, we proceed with the assumption of blocking processing as shown in Fig. 2. The blocking processing contrasts with the non-blocking processing in that no buffer exists between two adjacent threads. That is, a thread cannot pass the processed packet to its successor if the successor is busy. Since normally the packet processing overhead, including computation and memory access, is fairly distributed among threads, this simplified assumption has limited influence on the correctness of the model while considerably reducing the state space.

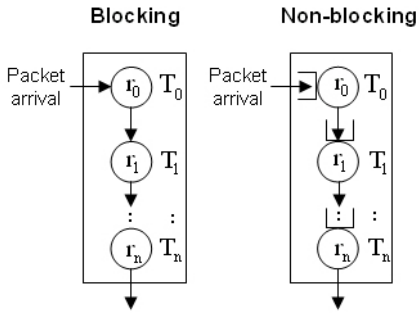


Fig. 2. The blocking and non-blocking packet processing schemes. A thread T_t accesses memory with rate r_t during the processing.

3.1. State Definition and State Space Determination

Our model considers *I* processors, each of which contains *J* threads, and aims to characterize the behaviors of processors, threads and memory. To do that, we need to clarify possible activities, i.e. *status transitions*, of a thread. They are depicted in Fig. 3 and elaborated below. When a packet arrives at an *idle* thread, the thread either enters the *ready* queue of the processor waiting for execution, or enters the *active* status if no thread is currently *active*. Sometimes it issues a *memory access* to, for instance, perform table lookups and manipulate packet descriptors. Once serviced it re-enters the ready queue, or goes directly back to execution if the ready queue is empty.

Normally, the thread becomes idle again after the packet is processed and passed to the succeeding thread. Nonetheless, it may get stuck and enter the *finished* status if the succeeding thread is busy with a packet.

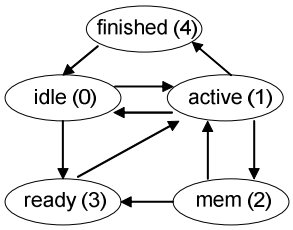


Fig. 3. Status transitions of a thread.

According to the above descriptions we can formally define a state of the system as

$S = (s_{0,0} ... s_{0,j} ... s_{i,j}),\ 0 \le i < I \text{ and } 0 \le j < J$,

where

$s_{i,j} \in \{0 : idle, 1 : active, 2 : mem, 3 : ready, 4 : finished\}$

represents the status of $T_{i,j}$, the *j*th thread in processor *i*. Furthermore we define $S(k) = \{s_{i,j} \mid s_{i,j} = k, 0 \le i < I \text{ and } 0 \le j < J\}$, so that the number of executing processors and number of accesses in the memory system equal to $|S(1)|$ and $|S(2)|$, respectively. We also define $h(i) = \{s_{i,j} \mid s_{i,j} = 2, 0 \le j < J\}$ so that the number of queued memory accesses of processor *i* is denoted by $|h(i)|$. Besides, the *RSS* (Random Selection for Service), rather than the FIFO, is assumed as the queuing discipline for both memory and ready queues. This assumption further diminishes the state space by disregarding the ordering information in the queues, and is proven not to affect the correctness of the analytical result in section 4. Taking (*I*,*J*)=(2,2) as an example, the state space can be derived by excluding unreachable states exhibiting the following properties:

1. A processor has more than one active thread. For instance, $(1,1,0,0)$.
2. At least one ready thread but no active thread, such as $(2,3,0,0)$. One of the ready threads must enter the active status as long as the previous active thread completes its processing.
3. $s_{i,j} = 4$ while $s_{i,j+1} = 0,\ 0 \le j < J-1$. In this case $T_{i,j}$ must pass the packet immediately to $T_{i,j+1}$ rather than staying *finished*.
4. $s_{i,J-1} = 4$; similar to 3, $T_{i,J-1}$ must send out the packet directly.

3.2. Determination of the Status Transition Diagram and State Transition Matrix

We will need the state transition matrix in order to solve the model. To derive the matrix, however, we have to deal with the status transition rate diagram of threads since a state change occurs when one or more threads alter its status. By assuming the packet arrival rate for processor *i* as λ_i, memory access rate and service time of the *j*th thread in that processor as $r_{i,j}$ and $1/\mu_{i,j}$, memory service rate as *m*, and number of queued memory accesses from the processor as *h*, we can have the status transition rate diagram shown in Fig. 4. Notably the service rates, as well as the memory access rates, of threads having same thread index in all processors are set the same because of the homogeneity among those threads. That is, $\mu_{i,j} = \mu_j$ and $r_{i,j} = r_j$. Packet arrival rates for all processors, which are also homogeneous, are set to λ.

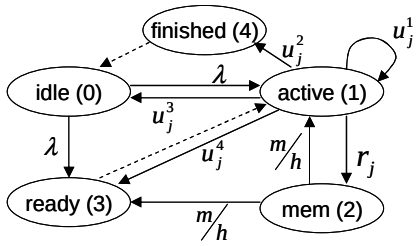


Fig. 4. Status transition rate diagram of $T_{i,j}$.

$\mu_j^k = 0 \text{ or } \frac{\mu_j}{n},\ n : \text{number of nonzero } \mu_j^k\text{'s}.$

Two additional transitions can be discovered out of Fig. 3 and shown in Fig. 4, the active to active and active to ready transitions. The former occurs when an active thread switches out and is then chosen again to process the packet from its finished predecessor; the latter is similar except that it is not chosen for execution but put into the ready queue.

Notice that two dotted status transitions in Fig. 4 do not have a rate because of being *follower transitions*. A status transition of a thread is regarded as a follower if it does not initiate a status transition but simply follows a certain *activator transition* which is launched by another thread actively. For example, a finished thread blocked by its successor can transit to the idle status (firing the follower transition) only after the successor finishes processing and passes down the packet (firing the activator transition). Another example is that a ready thread will never enter the active status unless a thread switches out from active. The state transitions and transition matrix can therefore be determined

according to the status transition diagram. Specifically, a state transition is considered valid if there exists only one *activation event* containing an activator transition and possibly a number of corresponding follower transitions.

3.3. Performance Estimation for the Analytical Model

The performance metrics that we are interested in from the analytical model include the processor and memory efficiencies. We can compute these measures from the stationary probability vector, π, for the Markov chain. The mean number of executing processors, which we call processing power (P_{power}), and the processor utilization, which we call processor efficiency ($P_{efficiency}$), are then calculated from the vector as

$$P_{power} = \sum_{S} (\pi(S) \times |S(1)|) \quad \text{, and} \tag{1}$$

$$P_{efficiency} = P_{power} / I \ . \tag{2}$$

Memory utilization, which we call memory efficiency ($M_{efficiency}$), number of memory accesses in memory system ($M_{accesses}$), and ready queue length of a processor (R_{length}) can be calculated as

$$M_{efficiency} = \sum_{S:\exists s_{i,j}=2} \pi(S), \tag{3}$$

$$M_{access} = \sum_{S} \pi(S) \times |S(2)|, \text{ and} \tag{4}$$

$$R_{length} = \left(\sum_{S} \pi(S) \times |S(3)| \right) / I \cdot \tag{5}$$

4. Simulation

Some tools have been available for simulating the CMP-based multithreaded architecture [14, 15]. Though accurate, they focus mainly on the low-level configuration such as cache structure and lack flexibility in thread allocation. In this section, we describe the construction of the simulation environment based on timed, colored Petri nets (CPNs) [16, 17]. It is used to validate the analytical model discussed in the previous section as well as to observe possible hints for future design.

4.1. Design of the Petri Net Based Simulation Environment

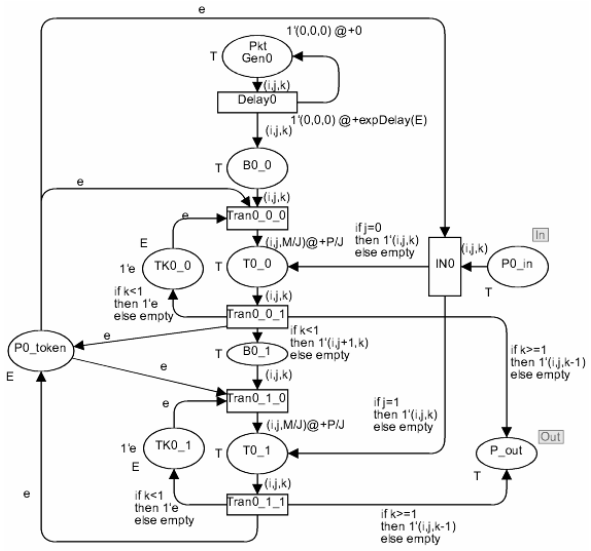
The key challenge in simulating memory queuing effect is that an outgoing memory access must go back to the thread where it is issued. For that purpose, we adopt the event-driven CPN-Tools [17] as our simulator. The features it supports, including the colored tokens, stochastic functions and hierarchical editing, provide efficiency in the construction of timed, colored Petri nets corresponding to our model. To give a general idea of the design of the Petri net based model, we use an example whose configuration of (*I*,*J*) is (1,2) shown in Fig. 5. Simulations for larger *I* and *J* are constructed in a similar way.

The sample Petri net implements the processor and memory subsystems shown in Fig 5(a) and 5(b), respectively, and works as following. A token is added as the initial marking in places such as the P0_token (for processor 0), TK0_0 and TK0_1 (for thread 0 and 1), Pkt_Gen0 (for packet generator), and Init (for memory). Among those tokens the one in Pkt_Gen0 is designed to be a colored token, which represents a packet and carries information about the processor index (*i*), thread index (*j*), and the number of memory accesses (*k*) the thread is obligated to perform to process the packet. The tokens of the others are simply non-colored ones.

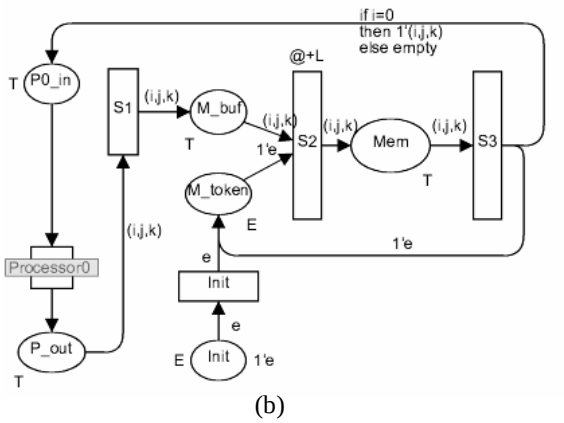
In the processor subsystem, the inter-arrival time of packets is exponentially distributed with mean *E* using the function *expDelay*, and the availability of a thread is dependent on whether a token is in both places of the processor and thread. When a packet arrives at B0_0, namely a colored token is fired by the transition Delay0, and if there is a token in both P0_token and TK0_0, the packet is admitted by consuming those three tokens and firing the transition Tran0_0_0. After that, the packet is processed for *P/J* computation cycles (active state) and *M/J* memory accesses are assigned to the thread by setting *k*= *M/J*, where *P* and *M* denote the numbers of computational instructions and memory accesses required to process a packet, respectively. The CPI (cycle per instruction) is assumed to be 1.

The memory access takes place by firing transitions Tran0_0_1 and S1 through P_out which is a common interface for all processors. The packet then enters the queue (M_buf) of the memory subsystem and gets serviced if no other is present. After a service time of *L* cycles (memory access state), the packet is passed back to T0_0 where it is issued according to the *i* and *j* in the token. The same procedure executes repeatedly until *k* becomes 0. The packet is passed to B0_1, waiting to be admitted by the next thread where operations similar to the above are carried out before leaving the system.

The simulation design differs from the analytical model in that the memory access rate and thread service rate are deterministic. The memory queue not shown in the above example is implemented in the M_buf using utilities of the CPN-Tools.



(a)



(b)

Fig. 5. An example hierarchical CPN describing (a) a processor containing two threads, and (b) the memory subsystem.

4.2. Model Validation by Simulations

The analytical model is validated by simulations. Our first observation is that, as presented in Table 2, the analytical results are mostly within 10% of the blocking simulation results. The discrepancy comes from the different assumptions between the model and the simulation. The former assumes non-deterministic behaviors in the instruction processing, memory access rate and memory service time, while the latter uses deterministic ones in order to be realistic. In fact, the discrepancy can be reduced to be less than 3% if all activities are presumed to be non-deterministic in the simulation. Second, the deviation further extends to be within 5-25% when comparing the blocking with the non-blocking simulation, meaning that the existence of buffer fairly influences the precision of the model. Due to the state space explosion of the analytical model, we focus on the simulation, specifically the non-blocking scheme which resembles real implementations.

Table 2. Validation of the analytical model against the blocking and non-blocking cases. The non-blocking case resembles real implementations.

(I, J)	Processor Utilization (%)				
	Ana.	Blocking	Non-Blocking	%(ana-B)	%(B-NB)
(1,4)	5.35	6.17	7.58	13.29	18.6
(2,2)	5.31	5.78	7.76	8.13	25.5
(2,3)	6.84	7.13	7.8	4.06	8.6
(2,4)	6.97	7.21	7.75	3.33	6.97
(3,2)	4.82	5.2	6.85	7.31	24.1
(4,2)	4.57	4.79	5.11	4.59	6.26

(a)

(I, J)	Memory Utilization (%)				
	Ana.	Blocking	Non-Blocking	%(ana-B)	%(B-NB)
(1,4)	26.56	29.31	36.56	9.38	19.83
(2,2)	51.1	56.57	74.15	9.67	23.7
(2,3)	67.77	70.63	74.92	4.05	5.72
(2,4)	68.45	71.26	74.58	3.94	4.45
(3,2)	68.78	77.11	99.84	10.8	22.76
(4,2)	87.43	99.84	99.99	4.14	8.78

(b)

4.3. Simulation Setup

Two networking applications, Simple Forwarding (SF) and DiffServ (DS), are involved in the simulations. The statistics of the computational and memory access instructions for handling a packet are configured according to [18] which uses a CMP-based multithreaded network processor. For simplicity, we assume that all memory accesses are of the same type, so the (*P*, *M*)s of the SF and DS are configured as (235, 12) and (555, 30).

Our goal is to investigate the relationship among processors, threads and memory banks. To do this, a term named *P-M ratio* is defined as

$$\frac{computational\ load}{mem\ access\ load} = \frac{\#\ of\ computational\ instructions}{\#\ of\ mem\ accesses \times latency\ per\ access},$$

and three sets of simulations are conducted: simulations with P-M ratio smaller than 1, close to 1, and larger than 1, respectively. A large (small) P-M ratio means the processor overhead is relatively higher (lower) than the memory's and is thought to be an unbalanced processor-memory combination, whereas a P-M ratio close to 1 is considered as a sensible one. In fact all networking applications can be categorized into these three aspects. Table 3 details the configurations of three different P-M ratios for the SF and DS. The memory service times of SRAM are set to 20 and 90 cycles, respectively, referring to the Intel IXP1200 and

IXP2400 [19] network processors. A memory service time of 5 cycles is also incorporated to simulate the case in which P-M ratio is larger than 1.

Table 3. Different kinds of P-M ratios: (a) smaller than 1, (b) close to 1, and (c) larger than 1. Memory access latencies are configured as those of the IXP1200 and IXP2400.

App.	Mem. access load	P-M ratio
SF	$12\times90=1080$	(a) $235/1080=0.217$
	$12\times20=240$	(b) $235/240=0.98\cong1$
	$12\times5=60$	(c) $235/60=3.92$
DS	$30\times90=2700$	(a) $555/2700=0.205$
	$30\times20=600$	(b) $555/600=0.925\cong1$
	$30\times5=150$	(c) $555/150=3.7$

4.3.1. Simulations with three P-M Ratios

Simulations with a P-M Ratio Smaller Than 1

Figure 6 shows the results of the simulations with P-M ratios smaller than 1 taken from Table 3. Apparently the memory access overhead is relatively so large that the processor efficiency is low and only two threads are enough to utilize the memory. The SF and DS have similar processor and memory utilizations because of similar P-M ratios.

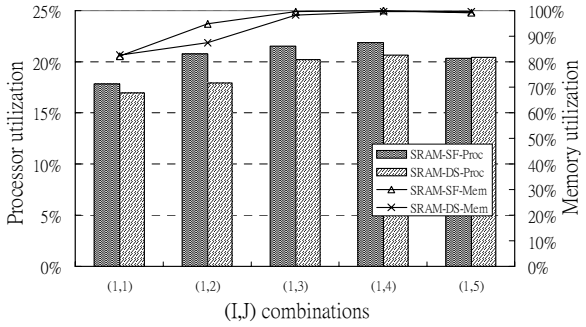


Fig. 6. Processor and memory utilizations for DS and SF with different numbers of threads. The memory service time is 90 cycles.

Simulations with a P-M Ratio Close to 1

Figure 7 shows the simulation results using a P-M ratio close to 1. SDRAM, another popular memory architecture in addition to the SRAM with a service time of 40 cycles is involved for comparison. From the figure we can see that for SRAM-SF and SRAM-DS the utilizations of both processor and memory are similar because the ratios are close to 1. Moreover, the benefit of utilizing memory from adding threads, taking the SDRAM-SF as an example, becomes less obvious as the memory utilization exceeds 90%. This observation also suggests that J=5 is most appropriate for applications with a P-M ratio close to 1, since the memory utilization of the SRAM-SF has reached 90% when J is 5 and the benefit from adding extra thread is limited. Specifically, we can further assert that J <5 is appropriate for the VPN (Virtual Private Network) processing while J >5 for the Intrusion Detection and Prevention as well as the Anti-Virus. The former has more computational operations whereas the latter two have greater memory access overhead, than the SF and DS.

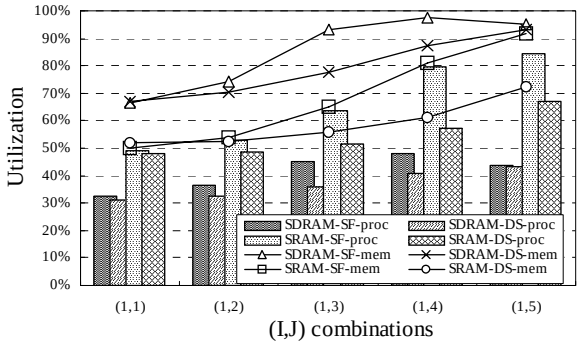


Fig. 7. Processor and memory utilizations for different numbers of threads.

Simulations with a P-M Ratio Larger Than 1

The memory service time is assumed to be 5 cycles so that memory overhead is relatively less than that of the processor. The corresponding P-M ratios are $235/(5\times12)\cong3.9$ and $555/(5\times30)\cong3.7$, respectively for SF and DS. The memory sustains the access load until four processors are incorporated for both SF and DS. Interestingly, though memory is apparently not a bottleneck when I=1 and 2, the processor is not fully utilized as shown in Fig. 8. This suggests that the J, which could lead to the low processor utilization, must be carefully estimated before using a fast memory module. Furthermore, the fifth processor contributes limitedly to utilizing the memory while resulting in low processor efficiency, implying that J, rather than I, should be increased to 4 when (I,J)=(4,3).

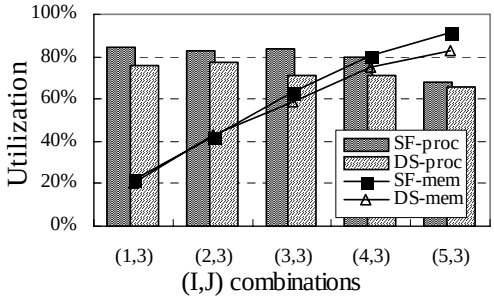


Fig. 8. Processor and memory efficiencies for different Is.

Discussions

The processing overhead of a packet is determined by the software, i.e. application, and hardware specifications, in which the former affects the number of computational and memory access operations while the latter determines the duration of each operation. Since the processing time is measured in *cycles* instead of normal time scales (ex: μsec), the metric of P-M ratio is independent of any single specification but their relative overhead; so are the discovered observations. Furthermore, examination on one P-M ratio can be applied to various software-hardware combinations. Therefore, rather than by involving several applications to gain software/hardware -dependent observations which is frequently unfeasible due to significant implementation effort, this approach derives general ideas by classifying all combinations into three aspects, namely smaller than one, close to one and larger than one.

With the investigation on these aspects and results from one of our previous studies [20], a projection can be estimated between P-M ratios and their corresponding *J*, as shown in Fig. 10. It is interesting to see that the demand for *J* rapidly lessens as the P-M ratio increases to 0.066, and slowly decreases afterwards. Notably the projection should be ladder-like in practice since *J* is always an integer.

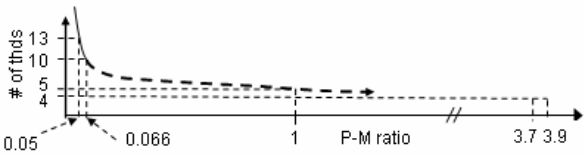


Fig. 9. Projection on the P-M ratios and the corresponding *J*'s. 10 and 13 are estimated for *J* when practically executing the Intrusion Detection and Prevention application with the Aho-Corasick and Wu-Manber algorithms, respectively.

4.3.2. Solutions for the Memory Bottleneck

Memory usually becomes the bottleneck because of not only the nature of the application but the speed gap between processor and memory. To tackle the problem, three common solutions are investigated and compared: enlarging the cache size for better hit ratio; adopting a memory access efficient algorithm, and adding more memory banks. Figure 10 compares the effectiveness of the solutions for the DS when (*I*,*J*)=(5,5) and *L*=20. The hit ratio is assumed to be 16.6% and 33.3%, respectively, by reducing the number of memory accesses from 30 to 25 and 20. As for the memory access efficient algorithm, we proceed by supposing a *classification algorithm* having memory accesses 50% less (from 10 to 5 accesses) while computational instructions 100% more (from 160 to 320 instructions) than the original algorithm, i.e. (*P*,*M*) from (555,30) to (715,25). The idea is that more computational instructions are usually traded for less memory accesses. We consider the effect of multiple banks by employing two banks and looking into two situations in which memory accesses are (1) evenly distributed and (2) distributed with ratios of 1:2 and 1:4. The cause of the second situation is the *data structure* and the nature of the application or the algorithm. For example, in a pattern matching application using the classic Aho-Corasick algorithm [21], it is unlikely to split the goto table evenly into memory banks, resulting in unbalanced memory access locality. Even if it is possible, the problem remains since the matching frequently returns to the root state stored in a certain bank.

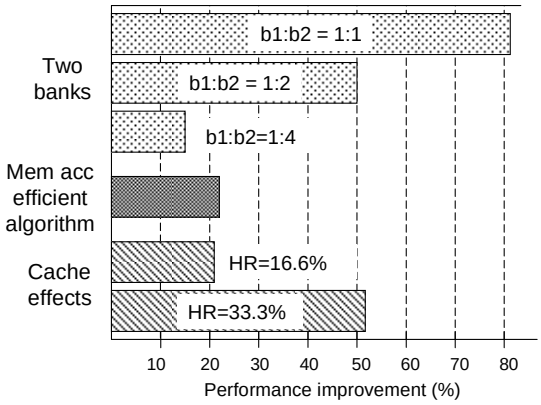


Fig. 10. Performance improvement from the three solutions with respect to (*I*,*J*)=(5,5) performing the DS. The hit ratio of 16.6% and 33.3% are simulated by using (*P*, *M*)=(555,25) and (555,20), while (715,25) is designed to mimic a system with a memory access efficient classification algorithm. Ratios of 1:1, 1:2 and 1:4 are investigated for the two-bank case.

From the figure we can see that with a hit ratio of 16.6%, an improvement of 21% can be obtained. The improvement advances to 51.5%, 2.5 times of that of 16.6% ratio, for a hit ratio of 33.3%. The benefit from a memory access efficient algorithm is 21.5%, similar to the one with 16.6% hit ratio, despite the increased number of computational instructions. The performance gain is best when introducing another memory bank. However, it degrades from 81% to 15% as the distribution of memory accesses becomes unbalanced.

5. Conclusions and Future Work

In this work, we try to derive possible design implications for CMP based multithreaded network processors by developing a preliminary analytical model as well as simulations based on the timed, colored Petri net. To date, this work is the first research that practically models *I* processors and *J* threads per processor based on the thread allocation discussions and queuing effect considerations for memory and ready queues.

Although the analytical model is verified to have similar behavior to the non-blocking simulation which quite resembles real implementations, we focus on the latter in order to have precise observations. The concept of P-M ratio, which is intended to cover general networking applications, is introduced and investigated; a projection is estimated between P-M ratios and the corresponding appropriate number of threads in a processor. It is found that the demand for *J* rapidly lessens as the P-M ratio increases to 0.066, and slowly decreases afterwards. *Observations from a certain P-M ratio can be applied to various software-hardware combinations having the same ratio.*

.As to solving the memory bottleneck resulted from small P-M ratio, adding memory banks best improves the performance, though the effectiveness depends heavily on the data structure of the application/algorithm.

Reference

[1] S. Kapil, H. McGhan, and J. Lawrendra, "A Chip Multithreaded Processor for Network-Facing Workloads," IEEE Micro, vol. 24, no. 2, 2004.

[2] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, "Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design," in Proc. of USENIX'05, April 2005.

[3] V. Ramamurthi, J. McCollum, C. Ostler, and K.S. Chatha, "System Level Methodology for Programming CMP based Multi-threaded Network Processor Architectures," in Proc. of International Symposium on VLSI (ISVLSI), May 2005.

[4] R. S.-B., D. Culler, and T. Eicken, "Analysis of multithreaded architectures for parallel computing," in Proc. of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, 1990.

[5] S.S. Nemawarkar, R. Govindarajan, G.R. Gao, and V.K. Agarwal, "Analysis of Multithreaded Multiprocessor Architectures with Distributed Shared Memory", in the Fifth IEEE Symposium on Parallel and Distributed Processing, Dallas, pp.114-121, 1993.

[6] M. Franklin and T. Wolf, "A Network Processor Performance and Design Model with Benchmark Parameterization," in *Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture*, 2002.

[7] T. Wolf and J.S. Turner, "Design Issues for High-Performance Active Routers," *IEEE JSAC*, vol. 19, no. 3, 2001.

[8] M. Gries, C. Kulkarni, C. Sauer, and K. Keutzer, "Comparing Analytical Modeling with Simulation for Network Processors: A Case Study," in *Proc. of the Design, Automation, and Test in Europe (DATE)*, 2003.

[9] P. Crowley, M. Fiuczynski, and J.-L. Baer, "On the Performance of Multithreaded Architectures for Network Processors," *UW Technical Report*, 2001.

[10] P. Crowley and J.-L. Baer, "A Modeling Framework for Network Processor Systems," in *Network Processor Workshop* in conjunction with *Eighth International Symposium on High Performance Computer Architecture*, 2002.

[11] W. Bux, W.E. Denzel, T. Engbersen, A. Herkersdorf, and R.P. Luijten, "Technologies and Building Blocks for Fast Packet Forwarding," *IEEE Communications Magazine*, January 2001.

[12] R. Ramaswamy and T. Wolf, "PacketBench: A Tool for Workload Characterization of Network Processing," in *Proc. of 6th IEEE Annual Workshop on Workload Characterization*, 2003.

[13] Intel IXP2400 network processor, http://www.intel.com/design/network/products/npfamily/.

[14] D. Nussbaum, A. Fedorova, C. Small, "An Overview of the Sam CMT Simulator Kit," *Technical Report of Sun microsystems*, June 2004.

[15] J.D. Davis, C. Fu, and J. Laudon, "The RASE (Rapid, Accurate Simulation Environment) for Chip Multiprocessors," in *Proc. of Workshop on Design, Architecture and Simulation of Chip Multiprocessors*, November 2005.

[16] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proc. of the IEEE*, vol. 77, no. 4, 1989.

[17] A.V. Ratzer et al., "CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets," in *Proc. of the International Conference on Applications and Theory of Petri Nets*, 2003.

[18] S. Lakshmanamurthy, K. Y. Liu, Y. Pun, L. Huston, and U. Naik, "Network Processor Performance Analysis Methodology," *Intel Technology Journal*, vol. 6 issue 3, 2002.

[19] D.E. Comer, "Network Systems Design using Network Processors," p. 282, Prentice Hall, 2004.

[20] Y.-N. Lin, Y.-C. Chang, Y.-D. Lin, and Y.-C. Lai, "Resource Allocation in Network Processors for Memory Access Intensive Applications," *Journal of Systems and Software*, Vol. 80, Issue 7, July 2007.

[21] A. Aho and M. Corasick, "Fast Pattern Matching: an Aid to Bibliographic Search," *Commun. ACM*, 18(6):333-340, June 1975.