Packet Processing with Blocking For Bursty Traffic on Multi-thread Network Processor

Yeim-Kuan Chang and Fang-Chen Kuo
Department of Computer Science and Information Engineering
National Cheng Kung University
701 Tainan, Taiwan R.O.C.
ykchang@mail.ncku.edu.tw
p7895107@mail.ncku.edu.tw

***Abstract*—It is well-known that there are bursty accesses in network traffic. It means a burst of packets with the same meaningful headers are usually received by routers at the same time. With such traffic, routers usually perform the same computations and access the same memory location repeatedly. To utilize this characteristic of network traffic, many cache schemes are proposed to deal with the bursty access patterns. However, in the multi-thread network processor based routers, the existing cache schemes will not suit to the bursty traffic. Since all threads may all deal with the packets with the same headers, if the former threads do not update the cache entries yet, the subsequent threads still have to repeat the computations due to the cache miss.**

In this paper, we propose a cache scheme called B-cache for the multi-thread network processors. B-cache blocks the subsequent threads from doing the same computations which are being processed by former thread. By applying B-cache, any packet processing tasks with high locality characteristic, such as IP address lookup, packet classification, and intrusion detection, can avoid the duplicate computations and hence achieve a better packet processing rate. We implement the proposed B-cache scheme on Intel IXP2400 network processor, the experimental results shows that our B-cache scheme can achieves the line speed of Intel IXP2400.

Keywords-multi-thread; network processor; cache; and Intel IXP2400

I. INTRODUCTION

In the multi-thread network processor, such as Intel IXP2400 [4], a single "*Micro Engine*" (ME) is the basic execution unit to a packet processing task (e.g., IP address lookup, packet classification, intrusion detection, etc.). A single ME of IXP2400 has eight threads which can be executed concurrently. Thus, in order to increase the performance, a single ME can do the packet processing task for at most eight different packets concurrently. In packet processing tasks, we may only focus on the subset of header fields (i.e. meaningful header). On the other hand, the network traffic usually appears with the burst pattern (i.e., a burst of packets with the same meaningful header are arriving at the same time), lots of packets need to be treated as the same way. For example, in IP address lookup, if the destination addresses of the incoming packets are the same, these packets should be forwarded to the same next hop. With the burst traffic, it will cause the increases of the duplicate computations and the number of unnecessary memory accesses. To avoid this wasted overhead, we may implement a naive cache for the packet processing task. The naive cache caches the results of the packet processing task for the former packets. For the subsequent packets with the same meaningful header, the cache hits can avoid the duplicate computations and hence increase the efficiency.

However, the naive cache described above may still not suit to the burst traffics. Since the traffics arrive in burst, all threads of a ME may all deal with the packets with the same meaningful header. If the former thread does not update the cache entry yet, the subsequent threads with the same meaningful header will still have to repeat computations due to the cache miss. This overhead is proportional to the number of MEs we allocate to the task. In other words, the more MEs we allocate to the packet processing task the more inefficiency we suffer. This paper is mainly focusing on solve the above problem. We propose a cache scheme called B-cache (B stands for blocking). B-cache blocks the threads which process the non-first-of-the-flow packets (i.e., subsequent packets) from repeat the same packet processing task as the former thread. Hence, the duplicate computation can be avoided and the higher packet processing rate can be achieved.

The rest of paper is organized as follows. Section II presents the related works. The proposed B-cache is introduced in Section III. The implementation issues of B-cache and its several variants are described in Section IV. An extensive performance results are shown in Section V. Finally, we conclude the paper in Section VI.

II. RELATED WORK

A. Cache Design for Packet Processing on Network Processor

Several studies focused on cache schemes for the network processor architecture. In [6] and [8], cache mechanisms are evaluated on Intel IXP1200 network processor. [6] is focused on the latency hiding techniques that the effect of multi-thread and cache are considered separately. On the other hand, [8] focus on the effect of cache to the different packet processing tasks. More parameters of cache are considered in the study.

Authors of [1] proposed the digest cache to increase the performance of packet classification. Different to traditional cache which stores the complete tag, the scheme stores only a hash of the tag. Thus, with the scheme, larger size of cache can be supported. Although the scheme can be used as independent cache, it can acts as the initial filter of exact match cache. The two levels cache architecture can solve the mismatched problem results from not to store the complete tag.

A hybrid cache scheme for network processor was proposed in [7]. Packets with the same source address, destination address, source port, destination port and protocol are said they belong to the same flow. With a traditional cache which cached the result of packet processing, the authors also proposed a cache which cached the additional information shared by the packets belong to the same flow. The authors proposed the cache which cached such information together in one cache entry to utilize the spatial locality.

This paper focus on avoid the problem which several threads duplicate computing to the packets belong to the same flow before the cache entry is updated by one of them. The problem will occur in multi-thread network processor environment. We try to delay the processing of such packets using the proposed scheme. As the redundant memory access and computing can be reduced, higher throughput can be achieved.

B. Implemented Packet Processing Scheme

The proposed cache scheme is suitable for any packet processing tasks with high locality characteristic. In this paper, we choose the HBSPC (hierarchical binary prefix search) [3] packet classification scheme for the evaluated packet processing task. Other well-known packet classification schemes can be found in [11]. Basically, [3] is extended from the ip lookup scheme BPS (binary prefix search) [2]. HBSPC use hierarchical structure to handle typical 5-dimensional rule tables. The two prefix fields of the rule tables are sorted and stored in the arrays which become the first and second level of the hierarchical structure. The last three fields are stored in the linked list pointed by the second level of the hierarchical structure.

The searching operation of HBSPC is as follows: The first step is to binary search the whole first level of hierarchical structure to find the LMP (longest matching prefix). The LMP has stored the information of search space of next level structure. With that, we can again binary search in the second level to find the LMP in the second level. Again, with the LMP, we can obtain the searching space in the third level structure. Finally, we can linear search the linked list to find the highest priority rule for the result.

To simplify the implementation, the HBSPC mentioned here is the basic version. The author of HBSPC has also proposed the improved version which requires less memory. The further detail can be found in [3].

III. PROPOSED CACHE SCHEMES

Network traffic has the busty access pattern. It is easy to implement a naïve cache scheme to handle such traffic to improve the overall performance. For an entry of a naïve cache, we at least have two main fields: **tag** and **result**. If the corresponding fields of subsequently packets matched the cached tag of the former packet, then the previous packet processing result, can be returned directly without further compute again. It is possible that the field tag is composed of several sub-fields. Obviously, the contents of the two fields should be different for different packet processing tasks. For ip lookup problem, the tag is the destination address of the packet while the result is the next-hop that the packet should forward to. For 5-dimensional packet classification problem, the field tags are source address, destination address, source port, destination port, and the protocol while the field result is the action which the packet should be treated.

In this paper, we don't focus the cache algorithm such as cache replacement policies or the degree of set associative. Thus the proposed B-Cache is an extension of the naïve cache. For the convenient, we will use the direct-mapped cache design through this paper. It is still easy to extend the design in this paper to implement different cache algorithms.

The main idea of this paper is that we hope to process the packet of the busty traffic only once. In other words, we only process the former packet and we block (i.e. delay) the subsequent packets of the busty traffic to be processed until the thread which processes the former packet has updated the cache. To achieve this, the B-Cache has the additional field **if_blocking**. The main function of this field is to indicate the former packet has been processing currently. The tag and result of the cache entry will be updated by the former thread after the processing is finished.

To indentify which packet is being processing, the former thread sets the field if_blocking with the tag of the packet and clears the field after the processing is finished. The field can be viewed as the second tag of the cache entry which enables other threads to identify what packet is being processing. The content to be stored in the field is related to the packet processing task. A thread can detect which kinds of packet (former or subsequent ones) it is processing currently when it checks the field for the first time. For a thread which process the subsequent packet, it can detect the processing to the former packet is finished by check the same field continually. Because our goal is that we only process the former packet, so the thread should block the subsequent packets to be processed

```
Packet_Processing_Procedure() {
    if tag is matched                // Case 1
        return result;
    else if if_blocking is set {     // Case 2
        blocking the packet until if_blocking is clear
        continue
    }
    else {                           // Case 3
        set if_blocking
        process the packet
        update tag and result
        clear if_blocking
    }
}
```

Figure 1. Searching Procedure of B-Cache

until the cache entry is updated. Figure 1 shows the packet processing procedure of the proposed B-Cache which described as follows:

Case 1: When a thread obtains a packet, it should checks if the corresponding fields of the packet have matched the tag within the cache entry first. If the result is “true” (i.e. cache hit), the cached result will return and the processing to that packet is finished (Line 02~03).

Case 2: If the cache is not hit, the thread should check if the field if_blocking has been set. If the result is “true”, it means the packet belongs to the subsequent packets. So, the thread should not process the packet immediately. Until block the packets to be processed for some period, the thread can re-check the field to determine if there is necessary to block the packet again. If the result is not, the thread is free to process the packet using the cached result (Line 05~06).

Case 3: If the cache is not hit and the field if_blocking is not set, the packet should belong to the former packet. It means that we have to process the packet actually. Before the real apply process operation to the packet, the thread needs to set field if_blocking using the tag of the packet to prevent subsequent packets go into the same step. After the processing is finished, the thread needs to update the tag and result of the cache entry to let subsequent packets can use the result directly. More important, the thread needs to clear the field if_blocking to unblock the blocked packets (Line 09~12).

We describe some of issues when we implement the proposed B-Cache and present the design we adopt in Section IV.

IV. Implementation Issue

A. *Intel IXP2400 Hardware Brief*

The Intel IXP2400 network processor has an ARM compatible XScale core and eight Micro Engines (ME) which can work in parallel or pipeline for processing packets in high speed. Each ME has eight threads which can execute concurrently to utilize the resource [4].

There are four kinds of memory units different in sizes and speeds that can be accessed by IXP2400 MEs. They are Local Memory, Scratchpad, SRAM and DRAM. Each ME has 640*4 Bytes Local Memory which is private to other MEs. Local Memory is the fastest memory unit. Each IXP2400 chip has 16 KB scratchpad which is the largest on-chip memory interface shared among MEs. DRAM is the largest and slowest memory interface of IXP2400. Although IXP2400 only supports one channel of DRAM, however, IXP2400 supports two channels of SRAM interfaces. The speed and size of SRAM are in the middle of Scratchpad and DRAM.

B. *Resource Allocation*

As IXP2400 has eight MEs, we allocate one for receiving packet, and another for transmitting packet. Tests we have done show that such setting is sufficient to achieve the maximum speed of IXP2400. For the reason, we are free to use the remained six MEs for implementing the evaluated cache schemes. Briefly, the forwarding rate will increase with the number of MEs we use, until the limitation of the scheme or maximum speed of IXP2400 is achieved.

We allocate SRAM for holding the data structure of evaluated packet processing scheme due to the memory requirement of the HBSPC. Besides, we use the scratchpad as scratch ring to implement inter-ME communication while the DRAM is used as packets buffer.

It is an issue of using which memory interface to hold the B-Cache. In our first tests, we place the B-Cache in the Local Memory with the reason it is the fastest memory of IXP2400. However, the size limitation makes it impossible to implement cache which size is larger than 128 entries. Thus, we decide to store the B-Cache in the SRAM. IXP2400 has two channels of SRAM interfaces. To balance the memory utilization, we store data structure of packet processing and B-Cache in the different channels of SRAM. There is another reason for us to store the cache in SRAM. The private property of Local Memory is not easy to implement the shared data structure among MEs. We believe that the shared cache outperforms than the distributed one.

C. *Cache Entry Design of B-Cache*

The B-Cache is an extension of the naïve cache with the additional field if_blocking which controls the procedure of packet processing. Because we won’t focus on the cache algorithm of the B-Cache, this section will focus on the design of if_blocking.

At first, we use one bit per field to present the processing state of the B-Cache entry. That is, if the bit is set, other packets should be delayed to be processed until the bit is clear. The design is easy to implement; however, it is possible that several packets which belong to the different burst traffic may be hashed to the same cache entry (i.e. B-Cache entry collision). We can’t detect the case with this cache design. In other words, when the field becomes unset, it is possible that the unblocked packet still miss-matched the cache which result in more tag comparisons.

The second design of the field is to store the tag of the packets which is being processed. This is the design described in the previous section. As the design, each cache entry will has two sets of the tag. The first tag is used with the cached result while the second tag is used to identify which packet is being processing currently. To achieve this, the second tag must be as large as the first one (i.e. all of the needed information are stored in the tag). It needs 104-bits to store the full tag for the 5-dimensional packet classification. Thus for the packet processing scheme used in this paper, it will need at least 208-bits per cache entry. That is too large to adopt in practice. As the tradeoff, we reference the design in [1] that the third design of the tag is obtained by hash all of the 104-bits into 32-bits content using the CRC function. The design will require less memory than the second one. Besides, the field can be checked in the least unit of SRAM access. The miss-classification problem described in [1] will not happen to us because the original tag will act the exact match cache. As the result, we adopt the third design in all of the tests which each B-Cache entry is 20-bytes.

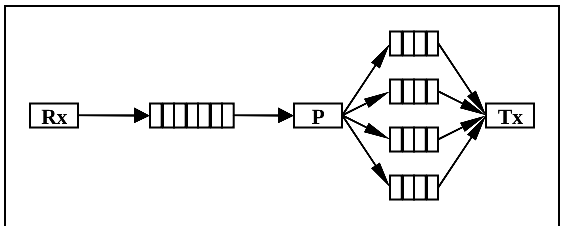


Figure 2. Architecture of Static Forward Project

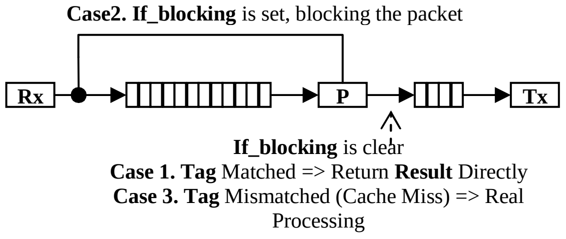
D. *Packet Blocking Procedure Design of B-Cache*

Another issue of B-Cache is how we done of blocking the packet from further processing. In this section, we will propose several architectures to handle this problem.

The proposed cache scheme is developed based on the ENP SDK [10] Static Forward project which is the example of an IXP2400 development board ENP-2611 [9]. In the architecture of the static forward project (Figure 2), the incoming packets will first receive by a ME (we note as *Rx* in this section) and after some simple operating by processing ME (note as *P*), the packets will be transmitted out of ENP-2611 by third ME (note as *Tx*). With the property of IXP2400, scratchpad can be programmed as FIFO scratch rings which can be used for inter-ME communication. In the static forward project, MEs exchange the information of packets through the scratch rings. We note the operation that writes the information of packet into the scratch ring as "enqueue the packet". In the other hand, the operation which reads the information of the packet from the scratch ring is noted as "dequeue the packet". As shown in Figure 2, there is a scratch ring between Rx and P ME. Besides, there are four scratch rings between P ME and Tx ME – one ring serves per physical port of ENP-2611. To reduce the space requirement of the figure, the four rings will be shown as one ring in the rest of the paper. With the same reason, although it is possible to use all of the remained six MEs for the packet processing at the same time, these figures will only show the case that using one ME.

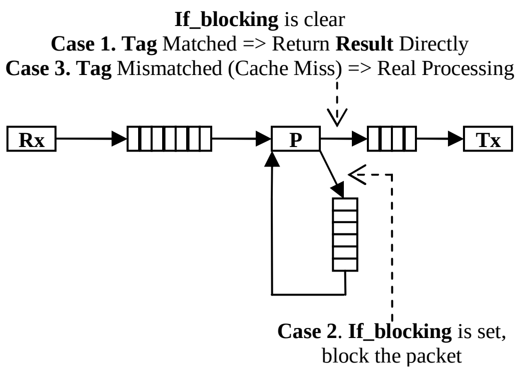
1) *Block Packet Using Original Scratch Ring*

The first design is based on the architecture of static forward project. As in Figure 3, when there is a packet which needs to be blocking, we enqueue the packet into the scratch ring which shared by Rx and P ME. This is the only difference between Figure 2 and 3. After the operation, the thread of P ME can freely dequeues another packet from the same scratch ring to handle with it. Because the scratch ring is shared by the Rx and P ME, lots of packets will be passed with the ring. With the higher latency of the ring it is difficult that the subsequent packets which come with the former packet has been dequeue again before the processing to the former packet is finished.



Case2. **If_blocking** is set, blocking the packet
Rx
P
Tx
If_blocking is clear
Case 1. Tag Matched => Return **Result** Directly
Case 3. Tag Mismatched (Cache Miss) => Real Processing

Figure 3. Block Packet using Original Scratch Ring



If_blocking is clear
Case 1. Tag Matched => Return **Result** Directly
Case 3. Tag Mismatched (Cache Miss) => Real Processing
Rx
P
Tx
Case 2. **If_blocking** is set, block the packet

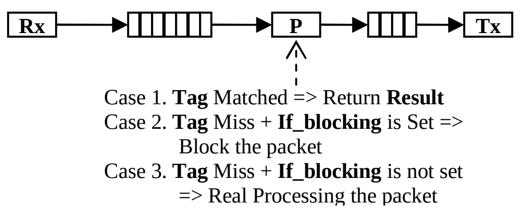
Figure 4. Block Packet using Additional Scratch Ring

When there is a collision of a specific B-Cache entry, which several packets are desired to update the same B-Cache entry, it will become a problem. Most of the packets will be cache miss due to the race condition. Even some of packets will become the former packets and have a chance to be processed; however, most of packets will be blocking and enqueue into scratch ring again and again. In our test, we can't make this design workable.

2) *Block Packet Using Additional Scratch Ring*

Different to the first design which use the original scratch ring to "buffer" the blocked packets, the second design (Figure 4) allocates another scratch ring (notes as *B-Scratch ring*) to temporary store the blocked packets. In the second design, the P ME has two scratch ring inputs need to be handled but not only one in the first design. There is an issue that "when" we should process "which" scratch rings as default. In our design, we will process the packet from the B-Scratch ring when previous result of packet processing belongs to case 1 and case 3 of the figure which usually indicates the processing of the former packet is done. If the result of previous processing belongs to case 2, we will enqueue the current packet into the B-Scratch ring and dequeue the new one from the ring shared by the Rx and P ME. After that, we will dequque the packet from the B-Scratch Ring until we process a packet which needs not to be blocking. In our tests, the second design is workable but performs worse than the naïve cache.

3) *Block Packet Without Using Scratch Ring*



Rx
P
Tx
Case 1. **Tag** Matched => Return **Result**
Case 2. **Tag** Miss + **If_blocking** is Set => Block the packet
Case 3. **Tag** Miss + **If_blocking** is not set => Real Processing the packet

Figure 5. Block Packet without using Scratch Ring

After implement and test the above two designs, we observe that the performance does not perform well. With some studies about the possible reasons, we believe that the performance is decreased due to the additional enqueue, dequeue operations and queuing delay. Further more, when the blocked packet is dequeued again, in our implementation, the thread which handle the packet need to access the DRAM again to obtain the packet headers for the packet processing task we implemented. The task we choose is a 5-dimensional packet classification scheme, that is, we need to access 104-bits to determine which operation we should take. We can't pass the header by using the scratch ring because it is too large, although the solution is workable for packet processing task such as ip lookup. To solve the problem, we propose the third design of blocking procedure.

As the Figure 5, the third scheme is almost looks like the original architecture of static forward project that there is no need of scratch ring to buffer the blocked packets. When a packet is needed to be blocked, we let the thread which handles the packet to swap out for a while using *ctx_swap* instruction. After the thread has a chance to execute again, we let the thread to check the field if_blocking again to determine if it is necessary to block the packet (i.e. swap itself) again. On the other hand, the procedure of the packet which needs not to be blocked is just the same as the previous designs. After the evaluation, we adopt this design and show the result in the Section V.

V. PERFORMANCE EVALUATION

A. *Simulation setup*

We simulate all of the codes with IXA SDK 3.5 Developer Workbench. The code is developed based on the ENP SDK 3.5 R4 Static Forward project. The original project was totally written in microcode which uses "receive-process-transmit" programming model. We rewrote the microcode of processing ME using MicroC [5]. Then, the packet processing scheme HBSPC and the proposed cache scheme (MicroC) were added for evaluating. We didn't change neither the receiving nor transmit related codes. We modified both the XScale and ME frequency from 400 MHz to 600 MHz in all tests which is the same as our ENP-2611.

The data structures used by the HBSPC are pre-computed using PC. We load the structures into Workbench by scripts. On the other hand, the structure for the proposed B-Cache is dynamically maintained by the processing ME. The structure of HBSPC is stored in channel 1 of SRAM while the proposed B-Cache is stored in the channel 0 of SRAM. Finally, we use the traces corresponding to the rule table to generate the packet streams used in the simulation.

B. *5-D Packet Classification Evaluation Settings*

We use ClassBench [12] to produce rule table *5D_5000* for the tests using the setting: "firewall". There are 4704 rules in this table. Other information can be found in Table I, and parameters to produce the table can be found in Table II.

To test the rule table, we use ClassBench again to produce two corresponding trace files *5000H* and *5000L*, where suffix indicate the locality. That is, 5000H indicates high locality trace while 5000L indicates low locality trace. We show the number of packets in each trace in Table III. The settings for

TABLE I. RULE TABLE FOR 5-D PACKET CLASSIFICATION

Rule Table	5D_5000
Number of Rules	4704
Number of different Destination Address	212
Number of different Source Address	65
Number of different Destination Port	49
Number of different Source Port	28
Number of different Protocol	9

TABLE II. PARAMETERS USED BY CLASSBENCH TO GENERATE THE RULE TABLE

db_generator -bc	<# of filetrs>	<smoothness>	<address scope>	<application scope>
5D_5000	5000	58	-0.5	-0.5

TABLE III. STATISTICS OF SIMULATION TRAFFICS

Name of traces	5000H	5000L
Locality	High	Low
# of Packets	48,099	48,113

TABLE IV. PARAMETERS USED BY CLASSBENCH TO GENERATE THE SIMULATION TRAFFICS

	<Pareto parameter a>	<Pareto parameter b>	<scale>
5000H	1	1	10
5000L	1	0.1	10

TABLE V THROUGHPUT OF HBSPC WITH THE PROPOSED CACHE SCHEMES (MPPS)

Scheme	**Cache Size**	**1ME**	**2ME**	**3ME**	**4ME**	**5ME**	**6ME**
High Locality Traffic (5000H)							
No-Cache	0	1.35	2.67	3.83	4.50	4.70	4.73
Naïve Cache	128	3.19	5.18	6.45	6.45	6.46	6.46
	256	3.24	5.30	6.45	6.45	6.46	6.46
	512	3.25	5.32	6.45	6.45	6.46	6.46
	1024	3.28	5.41	6.45	6.46	6.46	6.46
	2048	3.34	5.52	6.45	6.46	6.46	6.46
B-Cache	128	3.56	5.95	6.45	6.46	6.46	6.46
	256	3.56	6.00	6.45	6.46	6.46	6.46
	512	3.60	6.08	6.45	6.46	6.46	6.46
	1024	3.62	6.17	6.45	6.46	6.46	6.46
	2048	3.66	6.22	6.45	6.46	6.46	6.46
Low Locality Traffic (5000L)							
No-Cache	0	1.33	2.65	3.83	4.62	4.91	4.97
Naïve Cache	128	1.83	3.46	4.87	5.86	6.18	6.24
	256	1.84	3.49	4.94	5.92	6.24	6.30
	512	1.86	3.55	5.04	5.99	6.33	6.39
	1024	1.90	3.62	5.15	6.11	6.43	6.45
	2048	1.96	3.75	5.32	6.26	6.46	6.46
B-Cache	128	1.86	3.59	5.20	6.21	6.46	6.46
	256	1.88	3.64	5.26	6.29	6.46	6.46
	512	1.90	3.68	5.31	6.38	6.46	6.46
	1024	1.93	3.74	5.42	6.45	6.46	6.46
	2048	1.99	3.87	5.61	6.46	6.46	6.46

TABLE VI THE PERCENTAGE OF MEMORY ACCESS REDUCTION TO THE HBSPC STRUCTURE (%)

Cache Size	**128**	**256**	**512**	**1024**	**2048**
High Locality Traffic (5000H)					
Naïve	75.20	75.55	75.84	76.20	76.88
Proposed	84.37	84.33	84.44	84.64	85.01
Low Locality Traffic (5000L)					
Naïve	43.01	43.72	44.83	46.26	48.78
Proposed	46.94	47.59	48.59	49.92	52.18

producing these traces are shown in Table IV. All packets in above traces are 64 bytes. With the smallest size of Ethernet packets, we can observe the worst case performance of evaluated cache scheme.

C. Proposed B-Cache Evaluation

1) Throughput of HBSPC enhance by the B-Cache

The line speed of IXP2400 is OC-48, i.e. 2.5 Gbps. With the traffic which size of each packet is 64 bytes, IXP2400 should process 6.46 Million packets per second to achieve the maximum rate. We expect the evaluated cache scheme can achieve such throughput. Table V shows the throughput (number of million packet processes per second) of the proposed scheme using two kinds of traffic. The table compares the case with no-cache, naïve cache, and the proposed B-Cache. For the naïve cache and the B-Cache, we evaluate the cases which the cache size ranges from 128 to 2048. Different column of the table show the throughput using from one to six MEs for packet processing. We mark the case which achieves the maximum rate of IXP2400 whose background color as gray.

As the table, HBSPC can't achieve line rate without using the cache whether how many MEs are used. It can be seen that naïve cache improves the throughput of HBSPC for high locality traffic dramatically thus we can achieve the line speed when using more than three MEs. However, it is hard for the naïve Cache to achieve the same speed when using the low locality traffic. On the other hand, the goal can be achieved when we adopt the proposed B-Cache. The most important is that the throughput of HBSPC enhances with B-Cache becomes higher than the naïve cache when using high locality traffic. It is important that the B-Cache is just an additional cache scheme which solves the duplicate processing problem which other threads repeat handles the packet before the cache entry has been updated.

2) Reduction of memory access of HBSPC

It is obviously that the scheme requires less access to the memory performs better. Table VI compares the reduction of memory access to the HBSPC structure of B-Cache to the naïve cache. The table shows the reduction percentage with cache size from 128 to 2048 which only one ME is used for packet processing. The table presents the same trend as the Table V that the proposed B-Cache outperforms the naïve cache even with the low locality traces. The proposed B-Cache can reduce about 9% of memory access of packet processing with the evaluated high locality trace. For the low locality trace, the reduction is increased with the cache size. With the presented experiments, we believe that the proposed B-Cache can further enhance the throughput of packet processing in the multi-thread network processor environment.

VI. CONCLUSION

In this paper, we first implement a naïve cache scheme to improve the performance of HBSPC packet classification scheme. Basically, the cache scheme can solve the redundant processing problem to the busty traffic briefly. However, due to the Intel IXP2400 is a multi-thread processor that several threads are executes concurrently. It is possible that other threads will duplicate the processing of the packet due to cache miss before the cache is updated completely. In the case, the redundant processing to the packet will wasted computing power and involves additional memory access to the data structure. The B-Cache proposes in this paper solve this problem by blocking such packets from being processed until the cache is updated. With the proposed scheme, further throughput can be achieved on the IXP2400.

[1] Francis Chang, Wu-chang Feng, Wu-chi Feng, and Kang Li, "Efficient Packet Classification of Digest Caches", *Proc. of the Third Workshop on Network Processors & Applications (NP3)*, February 2004.

[2] Yeim-Kuan Chang, "Fast Binary and Multiway Prefix Searches for Packet Forwarding", *Computer Networks*, Volume 51, Issue 3, pp. 588-605, February 2007.

[3] Yeim-Kuan Chang, "Efficient Multidimensional Packet Classification with Fast Updates", Accepted in *IEEE Transactions on Computers.*

[4] Intel Corporation, "Intel® IXP2400 Network Processor Hardware Reference Manual", November 2003.

[5] Intel Corporation, "Intel® IXP2400/IXP2800 Network Processors Microengine C Language Support Reference Manual", November 2003.

[6] Zhen Liu, Hao Che, Kai Zheng, Shanzhen Chen, Chengchen Hu and Bin Liu, "A Trace Driven Comparison of Latency Hiding Techniques for Network Processors", *Proc. of the IEEE ICC 2006*, pp. 122-127, June 2006.

[7] Z. Liu, K. Zheng and B. Liu, "Hybrid cache architecture for high-speed packet processing", *Computers & Digital Techniques, IET*, Volume1, Issue 2, March 2007.

[8] Zhen Liu, Jia Yu, Xiaojun Wang, Bin Liu, and Laxmi Bhuyan, "Revisiting the Cache Effect on Multicore Multithreaded Network Processors", *Proc. of the IEEE DSD 2008*, pp. 317-324, Sepetmber 2008.

[9] RadiSys Corporation, "ENP-2611 Hardware Reference", August 2003.

[10] RadiSys Corporation, "ENP Software Development Kit Programmer's Guide", April 2004.

[11] David E. Taylor, "Survey and Taxonomy of Packet Classification Techniques", *ACM Computing Surveys*, Volume 37, Issue 3, pp. 238-275, September 2005.

[12] David E. Taylor and Jonathan S. Turner, "ClassBench: A Packet Classification Benchmark", *IEEE/ACM Transactions on Networking*, Volume 15, Issue 3, pp. 499-511, June 2007.