Towards Optimized Packet Processing for Multithreaded 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, p7895107}@mail.ncku.edu.tw

***Abstract*—With the evolution of the Internet, current routers need to support a variety of emerging network applications while the high packet processing rate is still guaranteed. As a result, the network processor has become a promising solution for network devices due to its computation capability and programming flexibility. However, developing the network applications on network processors is not easy. How to efficiently program multiple processing elements and utilize various memory modules as well as the hardware resources on network processors are always challenges. In this paper, we investigate several optimization issues and programming techniques that should be considered by the developers to achieve higher packet processing rate on network processors. We use an existing packet classification scheme called hierarchical binary prefix search (HBPS) [1] as the benchmark to test and evaluate these optimization techniques. The experiments conducted on Intel IXP2400 network processor show that the overall performance of HBPS can be improved about 42% while these techniques are adopted.**

***Index Terms*—network processor, Intel IXP2400, and packet classification.**

I. INTRODUCTION

To keep up with the rapid growth of network link rate as well as Internet traffic, current backbone Internet routers have to forward millions of packets per second at each port. In order to achieve such high packet processing rate, hardware devices, such as ASIC, FPGA, and TCAM are usually adopted in the design of current routers. However, the use of these hardware devices makes routers difficult to upgrade and support many new network applications. Moreover, hardware devices will consume too many electric power and board area. Thus, these kinds of hardware-based routers are only suitable to be deployed in the backbone where the fast packet processing rate is the only consideration.

Current Internet routers need to support a variety of emerging network applications while the high packet processing rate is still guaranteed. To fulfill the requirement of high packet processing rate, network processors (NP) [6,10,11,12,13] are introduced as a promising solution for building the network devices. The network processor is a programmable processor that contains multiple processing elements, different memory modules, and unique instruction set specially designed for dealing with the network applications. However, to develop the network applications on network processors is quite different from that on general purpose processors. How to allocate data in different memory modules, how to arrange different functions among multiple processing elements, and how to efficiently use the specially designed instructions are all programmers' overheads. If network application programmers do not exploit the resources and characteristics of network processors efficiently, the developed network applications may not achieve the ideal throughput as the programmers expect.

In this paper, we investigate many optimization issues and programming techniques that can significantly improve the performance of the network applications on network processors. Some of these techniques are based on the inherent characteristics of network processors and some of them are based on our empirical discoveries. We apply the investigated techniques to the packet classification algorithm on network processors. Packet classification is the essential building block for many emerging network applications, such as virtual private network (VPN), network intrusion detection system (NIDS), policy routing, traffic billing, and other value added services. With the emergence of these applications, packet classification plays a very important role in the design of current Internet routers. We use an existing packet classification scheme called hierarchical binary prefix search (HBPS) [1] as the benchmark to test and evaluate the optimization techniques investigated in this paper. The experiments are conducted on Intel IXP2400 network processor. The experimental results show that the overall performance of the benchmark can be improved significantly.

The rest of the paper is organized as follows. In section II, we briefly review the HBPS scheme which is used as the benchmark in this paper. In section III, we briefly introduce the architecture and characteristics of Intel IXP2400 network processor. In section IV and V, we investigate several optimization issues and programming techniques on network processors. The experimental results are shown in section VI and section VII concludes this paper.

II. RELATED WORK

A. *Packet Classification*

The problem of classifying the packets according to a set of rules into different classes or flows is known as packet classification. Usually, the rules are pre-defined and contain fixed number of fields. A rule is called a match if all the fields of the rule match the corresponding headers of the incoming packet. The 5-D packet classification problem considers the

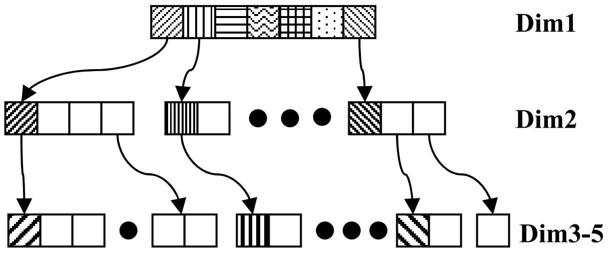

Figure 1. Example of HBPS.

source IP address, destination IP address, source port, destination port, and protocol number of the packet's header. The policies of comparison for these fields differ from each other make the packet classification problem more complicated. Each rule is associated with a priority. Usually, among all of the matched rules, the one with the highest priority will be returned as the result of packet classification. A complete packet classification scheme includes 1) pre-processing phase to build data structure and 2) search phase which classifies the packets by searching the data structure built. In this paper, we focus on the search process of the Hierarchical Binary Prefix Search (HBPS). More packet classification schemes can be found in [8].

B. Hierarchical Binary Prefix Search (*HBPS*)

HBPS [1] is a packet processing scheme for handling the 5-D packet classification problem. The data structure of HBPS can be viewed as a hierarchical structure of sorted array. An example of HBPS is showed in Figure 1.

HBPS uses the source address of the rules to build the first level of sorted array (dimension 1). Then, the remaining rules are leaf-pushed to the corresponding entries of dimension 1 array. The destination addresses of the pushed rules are used to build the second level of sorted array (dimension 2). After leaf-pushing the rules to the dimension 2 array once again, the source port, destination port, and protocol of the remaining rules are used to build the linked list sorted by the rule's priority.

```
01 HBPS_Search ( )
02 {
03    Variable Search_Space_D1, Search_Space_D2, \
       Search_Space_D35
04    Variable LPM_D1, LPM_D2, LPM_D35
05
06    Search_Space_D1 = 0..(Array_Size_D1-1)
07
08    // Search Dimension 1
09    LPM_D1 = Binary_Search_D1()
10     If( LPM_D1 != NULL)
11        Get Search_Space_D2 from structure of LPM_D1
12     Else No_Matched_Rule()
13
14    // Search Dimension 2
15    LPM_D2= Binary_Search_D2()
16    If( LPM_D2 != NULL)
17       Get Search_Space_D35 from structure of LPM_D2
18    Else No_Matched_Rule()
19
20    // Search Remained Dimension (D3-5)
21    LPM_D35=Linear_Search_D35()
22    If( LPM_D35 != NULL )
23       Get Matched Rule from structure of LPM_D35
24    Else No_Matched_Rule()
25 }
```

Figure 2. Pseudo Code of HBPS-Search.

```
01 Binary_Search_D1()
02 {
03    Variable Begin, End, Middle
04
05    Set_Search_Space( Begin, End, Search_Space_D1 )
06
07   While( Begin < End )
08    {
09       if ( (Begin==End) || ((Begin+1)==End ) )
10       {
11           Read Array_D1[Begin] into Prefix(Begin)
12           Read Array_D1[End] into Prefix(End)
13           Check if Prefix(Begin) Matched the Address
14           Check if Prefix(End) Matched the Address
15           Break
16       }
17       Middle = (Begin+End) >> 1
18       Half_the_Search_Space( Begin, End, Middle,
Higher_OR_Lower );
19    }
```

Figure 3. Pseudo Code of Binary-Search-D1.

Figure 2 shows the search codes corresponding to HBPS. The HBPS search process begins at the dimension 1 array. After binary searching (while-loop) the dimension 1 array to find the longest prefix match (LPM) (line 09), the search space in the next dimension (dimension 2) are obtained from the LPM entry (line 11). Following the same process done in dimension 1, the search space in dimension 3-5 can be obtained in dimension 2 (lines 15-18). The search process ends after linear search with the linked list in dimension 3-5 (lines 21-24).

Figure 3 shows the initial design of the function *Binary_Search_D1* used in line 09 of Figure 2. The search space [*Begin..End*] is obtained from the global variable *Search_Space_D1*. During searching in dimension 1, HBPS needs to find the longest prefix match. But in the case of general binary search process, it will abort if an exact matched occurs. Thus, it is possible that the sorted array is not fully traversed after a match is found. However, in the case of HBPS, the search algorithm need to traverse the whole array Array_D1 until only two prefixes remain in the search space. It makes the function *Binary_Search_D1* looks different (lines 07-17). The operation to find the LPM of the source address is applied to the remaining entries once the last two entries remain in the search space (lines 11-14). In the other cases, the search space will be halved. The remaining search space is determined by the comparison of the source address and the corresponding prefixes of the middle entry of the search space (lines 17-18).

We use the same design in function *Binary_Search_D2* used

```
01 Linear_Search_D35 ()
02 {
03    while( Search_Space_D35 )
04    {
05       Read_a_D35_Node
06       Check_Rule
07       if( Match ) break
08    }
09 }
```

Figure 4. Pseudo Code of Linear-Search-D35.

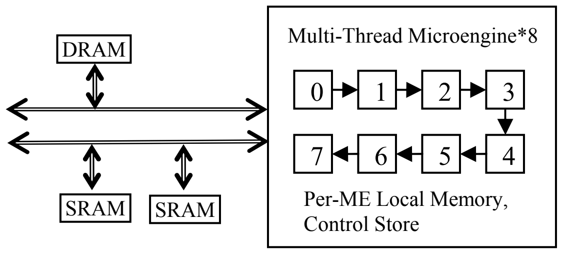
DRAM
Multi-Thread Microengine*8
0
1
2
3
7
6
5
4
SRAM
SRAM
Per-ME Local Memory, Control Store

Figure 5. IXP2400 component diagram.

in line 15 of Figure 2. On the other hand, the detail of the function *Linear_Search_D35* is shown in Figure 4. Basically, the search process in linked list is only aborted after the match occurs or all of the nodes of linked list are checked.

III. Hardware Architecture of Evaluated Network Processor

In this paper, Intel IXP2400 network processor [2] is chosen as the experimental platform due to its popularity in the network research area. Figure 5 shows the component diagram of Intel IXP2400. The processor has an ARM compatible XScale core (not shown in Figure 5) and eight Microengines (ME) which can execute in parallel or pipeline for processing packets in high speed. Each ME has its own control store which can store 4K 40-bits instructions. A microengine executes the instructions in the control store under the control of XScale.

Each ME has eight threads which execute concurrently to cover the latency of memory accesses. Once a thread issues a memory request, the thread needs to swap itself to let another thread to continue executing until the memory request is completed. Such mechanism prevents ME's from being idle.

IXP2400 can access four kinds of memory units which differ in terms of sizes and speeds. They are Local Memory, Scratchpad, SRAM, and DRAM. Each ME has 640*4 bytes Local Memory which can not be accessed by other microengines. Local Memory is the fastest memory unit. DRAM is the off-chip memory for buffering the content of the incoming packets. The 16 KB scratchpad can be programmed as multiple scratch rings. These scratch rings can be shared by all ME's. Besides, these scratch rings are the FIFO structure for ME's to exchange the handles and other information of the packets stored in DRAM. Further, IXP2400 supports two channels of SRAM. SRAM are used for storing the controlled data.

IV. Survey of NP Programming Techniques

Many optimization techniques can be applied for programming network processors. These techniques can be classified as follows.

A. NP Independent Techniques

The goal of the NP independent techniques reduces unnecessary computations. One way to achieve such goal is to reduce the instructions that are actually executed. For example, the common values that need to be computed online can be pre-computed and stored in a faster memory for lookup. Also, the authors in [4] suggested to inline the short functions which will be called frequently. The IXP Microengine C compiler has the directives *__inline* and *__forceinline* that we can use to guide the inline policy of the compiler.

B. NP Memory Dependent Techniques

The techniques listed here are mainly for reducing the latency of memory accesses. The overhead for accessing memory usually is the main reason for poor performance of packet processing tasks in routers. It is obvious that the scheme issuing less number of memory accesses will perform better. But it would be different if the latency for memory accesses can be overlapped.

- Using the fastest memory

The easiest solution of the above problem is to use the fastest memory interface to allocate the variables. However, the drawback is that the fastest memory is expensive and so its size is usually smaller than needed.

- Following the characteristics of the memory interfaces

Each memory interface works under its own characteristics. Those characteristics may make the previous techniques become advantageous. For example, each processing element (PE) of IXP2400 contains a memory private to each PE called *Local memory* as described above. Before accessing the local memory, one additional operation needed to be performed in advance is that one of the controlled registers needs to be set to point to the target address. The desired memory access can be performed in only three cycles after the above additional operation. However, once the controlled register is written, the local memory access can be improved with the self-increment pointer operation supported by IXP2400. With this mechanism, the overhead to set the controlled register can be avoided if the address of the data to be accessed is just sit at the next address of the current data. Thus, better performance can be obtained if such characteristics of specific memory have been considered.

- Issuing multiple requests at a time

Issuing several memory requests at a time instead of issuing the requests sequentially is a useful technique for hiding the latency of memory accesses. For example, this technique can be applied to lines 11-12 in Figure 3 because the addresses of variables which need to be accessed (Begin, End) are independent to each other. The modified design is shown in line 13 of Figure 6.

- Using wide word access (burst read/write)

Programmers are allowed to combine several memory requests to the contiguous memory addresses into a single multiple-words memory request. For example, the function *Linear_Search_D35* shown in Figure 4 is so designed to read one linked list node and check whether the rule is matched. In our implementation of HBPS, we map all of the nodes of the linked list into a block of contiguous memory locations. Due to above reason, multiple requests to contiguous linked list nodes can be combined into one request to reduce a number of memory accesses. As a result, the less is the number of memory requests issued, the lower is the overhead on the command bus.

C. NP instruction dependent techniques

The instruction dependent techniques are network processor dependent because no processors will implement the same instruction set. It's the programmers' responsibility to be

```
01 Binary_Search_D1_V2()
02 {
03-06 ……
07   While( (Begin+2) <= End )
08   {
09      Middle = (Begin+End) >> 1
10      Half_the_Search_Space( Begin, End, Middle, \
                                       Higher_OR_Lower )
11   }
12
13   Read Prefix Array_D1[Begin] and Array_D1[End] \
      into Prefix(Begin) and Prefix(End)
14   Check if Prefix(Begin) Matched the Address
15   Check if Prefix(End) Matched the Address
16 }
```

Figure 6. Pseudo Code of second design of Binary-Search-D1 (HBPS-V2)

familiar with the available instruction set, especially the bit manipulation instructions. For example, IXP2800 B0 compatible processor supports an instruction *pop_count* which can count the number of set bits in a 32-bit register. For the packet processing schemes which use bitmap compression techniques, such instruction will be useful. For example, Bitmap-RFC [5] utilizes this instruction to access the compressed data structure. Without the instruction, the operation can not perform efficiently. In [5], an alternative implementation using another instruction *FFS* [4] which is also available in IXP2400 processor has been proposed too. But it has been shown in [5] that this alternative implementation is less efficient than the original one. As a result, choosing the most suitable instruction is very important to design an efficient implementation for the packet processing tasks.

V. Techniques To Packet Processing

A. The Basic Design of HBPS

For convenience sake, we call our techniques for implementing HBPS described above as ***HBPS-V1***. Further evaluation results of the implementation can be found in Section VI.

B. The Second Design of HBPS

We observe that the while loop of *HBPS-V1* occupies a great percentage of the total executing time. Besides, such process will be used twice in the searching process of HBPS (dim1 & dim2). So we focus on decreasing the overhead of the while loop. After examining the codes in Figure 3, we observe that the IF statement (line 09) will be executed when we have to determine the relationship between variables *Begin* and *End*. However, the result of comparison is usually false. Also, the comparison to *Begin* and *End* in the While-Loop (line 07) is always executed but will never return the false result. While considering the binary search design (Figure 3) of *HBPS-V1*, the problems of the design we observed are:

- The WHILE-statement is redundant in all of the cases.

We use the statement (line 07) to determine the condition to abort the while loop. But the condition of the WHILE statement is loose than the IF statement (line 09). The IF statement will result in that the while loop is broken before the condition of the while statement becomes false.

- The return of IF-statement is false for most of the cases.

```
01 Binary-Search-D1-V3()
02 {
03-06 ……
07   if( address >= the prefix in the middle of the array )
08       Begin= (Begin+End ) >>1
09   else
10       End = (Begin + End ) >>1
11
12   While( (Begin+2) <= End )
13    {
14       Middle = (Begin+End) >> 1
15       Half_the_Search_Space( Begin, End, Middle,
Higher_OR_Lower )
16    }
17
18    Read Prefix Array_D1[Begin] and Array_D1[End] into \
19    Prefix(Begin) and Prefix(End)
20    Check if Prefix(Begin) Matched the Address
21    Check if Prefix(End) Matched the Address
22 }
```

Figure 7. Pseudo Code of the third design of Binary-Search-D1 (HBPS-V3)

Instructions need to be loaded into the instruction pipeline before executing. If the pre-loaded instructions are useless when the branch perdition fails, these instructions are needed to be cleared from the pipeline. Some machine cycles will be wasted before the instructions in the correct branch are loaded. According to the optimization techniques listed in [4], the compiler will generate the binary code with the decision when the default branch in the IF-statement is taken. That is, the instructions in the IF statement will be loaded into the pipeline. But the design in Figure 3 usually gets the false result and will lead to the pipeline abortion and thus decrease the performance.

Based on the observations described above, we change the design of *Binary_search_D1* as shown in Figure 3 to the second version shown in Figure 6. The changes are: 1) we combine the only comparison between *Begin* and *End* in line 7 and the two comparisons also between *Begin* and *End* in line 9 of Figure 3 into one comparison. 2) We move the statements for checking whether the address matches the prefixes or not out of the while-loop. With these two changes, the search process shown in Figure 6 becomes: 1) the binary search is performed on the prefix array first and the search space is finally reduced to only two prefixes (lines 07-11). 2) only one read operation on array to obtain Prefix (*Begin*) and Prefix (*End*) is performed (line 13). 3) the operation to determine which prefix or none of prefixes matches the address is performed (lines 14-15).

The design of Figure 6 is also suitable for the function *Binary_Search_D2* used in line 15 of Figure 2. As a result, we use the second design of binary search to replace both of the *Binary_Search_D1* and *Bianry_Search_D2* of the HBPS. We call the design as ***HBPS-V2***.

C. The Third Design of HBPS

The third design of HBPS is developed based on the following observations. The *control store (or called instruction store) [2]* which stores the instructions to be executed is under-utilized. Each control store of Intel IXP2400's ME can store up to 4K 40-bits instructions. But we observe that *HBPS-V1* and *HBPS-V2* only occupy 325 and 315 instructions which are much less than the 25% of the total control store.

Our goal of the third design is to embed some of the static data structures into the control store. Because the structure is hardcoded, the external memory access to the structure can be avoided. As a result, the speed of the packet processing will become faster. Obviously, the data structure to be hardcoded should be frequently accessed and are small enough to be kept in the control store.

We apply this technique to *HBPS-V2* by hardcoding the first level of array into control store. Figure 7 presents a possible modification. In the loop of binary search, a number of the middle entries of the search space are fetched and compared to the address. The next search space to be binary searched is determined by the relationship of the comparison. For example, in the first iteration, the middle entry of the search space [*Begin*..*End*] says that *middle* is fetched. After the comparison, the search sub-space will be either [*Begin*..*Middle*] or [*Middle*.. *End*]. For the two possible search sub-spaces, there are two possible middle entries. With the trend, the more time we un-loop the while loop, the more possible data structure we need to keep in the control store.

Figure 7 is the design corresponding to Figure 6 which just un-loop the while loop once. The statements (lines 12-16) are not removed because it is not fully un-looped. The time that the statement can be un-looped actually depends on the rule tables and the size of available the control store.

We apply the second technique to *HBPS-V2* to develop *HBPS-V3*. Because the size of available control store is still large enough, we hardcode the whole structure of the first level (dim1) array into the control store. We call the design as ***HBPS-V3***.

D. The Fourth Design of HBPS

After adopting more than three ME's for packet processing, the design *HBPS-V3* reaches its limitation. No further performance can be obtained by using more ME's. The fourth design ***HBPS-V4*** is based on the observation that the average length of the SRAM command bus FIFO is long. If the length of the bus is long, longer delay will be needed to complete the memory requests. According to [7], we just distribute the data structure to both of the SRAM channels to solve the problem.

Table I compares the statistics before and after applying this technique to the design. Before applying the technique, the average length of the command bus may exceed 5. But after moving some data structure to another channel of SRAM, the average length of the SRAM channel command bus is reduced.

Table I. The analysis of SRAM Read Command Bus FIFO of HBPS-V3 and HBPS-V4

	1ME	**2ME**	**3ME**	**4ME**	**5ME**	**6ME**
HBPS-V3 (Structure in SRAM Channel #1 only)						
Channel #0	0.01	0.03	0.04	0.04	0.04	0.04
Channel #1	0.23	1.15	3.88	5.81	6.31	6.44
HBPS-V4 (Structure in SRAM Channel #0,1)						
Channel #0	0.16	0.33	0.53	0.71	0.80	0.84
Channel #1	0.06	0.34	1.50	3.13	4.05	4.39

VI. Performance Evaluation

A. Simulation Setting

We survey several techniques in the previous section. Besides, we also apply some of them to enhance the design of HBPS. For now, we will evaluate these designs to discover how much enhancement can be obtained from these optimization techniques. In this paper, we simulate all of the evaluated schemes under Developer Workbench which comes from IXA SDK 3.51 [3]. All of the evaluated codes are written in MicroC [4].

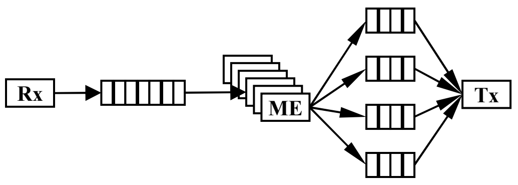

Figure 8. Architecture of the Evaluation Platform

Our simulation refers to the setting of Radisys ENP-2611 evaluation board [6]. The board contains one Intel IXP2400 network processor. For ENP-2611, all of the processing elements include XScale and MEs are executing at 600 MHz. With the eight ME's of IXP2400, we allocate one ME for receiving packets while another one for transmitting packets. The remaining six MEs can be allocated for evaluating the HBPS (Figure 8). But we mainly focus on the case that using one ME for packet classification.

B. Rule and Trace for Packet Classification

To benchmark these techniques, we use ClassBench [9] to generate a rule table with 4,704 rules and a trace that corresponds to this table. The trace contains 48099 headers. Throughout this paper, all of the experiments are evaluated based on the same rule table and trace.

To simulate the search process of HBPS without constructing the needed data structure first, we pre-built and convert the corresponding data structure of HBPS to Workbench compatible scripts. In the case of *HBPS-V1*, *HBPS-V2*, and *HBPS-V3*, the scripts are loaded to channel 1 of SRAM before the simulation begins. But in the case of *HBPS-V4*, the scripts are distributed to the both channels of SRAM.

The trace generated by ClassBench is equally divided to three parts. The three partial traces are converted to the stream format compatible to Workbench. The three streams are used for the three receive ports supported by ENP-2611. Thus, the forwarding rate of packet processing is obtained after all of the packets are transmitted.

C. Evaluation of Different Designs

We use several metrics to compare four implementations of HBPS.

- Forwarding Rate

The first part of Table II compares the forwarding rates of four designs of HBPS while only one ME is considered for packet processing. As a result, *HBPS-V2* is faster than

Table II. The analysis of the four implementation of HBPS (1ME)

	V1	V2	V3	V4
Forwarding Rate (Mbps)	691.03	841.83	985.38	987.05
Speed-Up to HBPS-V1	100 %	121.82 %	142.60 %	142.84 %
Pipeline Abort Rate	29.86 %	23.13 %	21.09 %	21.13 %
Number of Instructions	325	313	966	966
Utilization of Control Store	7.93 %	7.64 %	23.58 %	23.58 %
Avg. Number of Needed SRAM Access	21.46	21.46	13.68	13.68

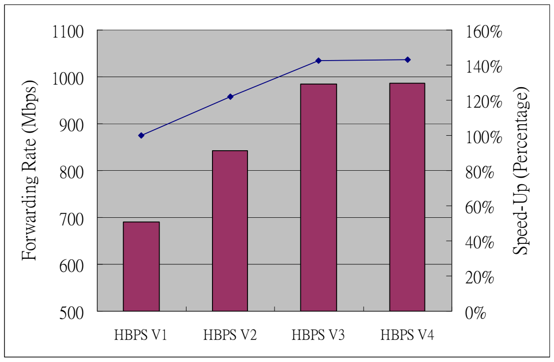

Figure 9. Forwarding rates and speed-ups.

HBPS-V1 while the *HBPS-V4* is the fastest one among all four designs. It can be observed that *HBPS-V4* is 42 % faster than *HBPS-V1*. Figure 9 shows the speed-up of the forwarding rates relative to *HBPS-V1*.

- Percentage of instruction pipeline aborted

The second part of Table II compares the rates of instruction pipeline aborted. The abort rate of *HBPS-V1* is 29.86 % and the rate of *HBPS-V2* becomes a lower rate of 23.13 %. It means that about 29.86 % of total simulation time that *HBPS-V1* does not execute because the instruction pipeline is aborted. With the newer design of the while loop, *HBPS-V2* fixes the problem. With the technique, *HBPS-V2* becomes 21 % faster than *HBPS-V1* while no additional mechanisms are adopted.

- Utilization of the control store

The third part of Table II compares the number of instruction (*uword*) used for packet processing ME. *HBPS-V3* and *HBPS-V4* have the highest utilization among these four designs because we hardcode the data structure of the first level array into control store. The instruction size of *HBPS-V3 and HBPS-V4* is the same because the only difference between these two designs is the memory interface that holds the data structure.

The fourth part of Table II compares the needed number of memory requests to access the HBPS structure which is stored in SRAM. By considering the information with the used instructions, we can observe that hardcoded technique reduces about 7.78 times of memory accesses but needs additional 653 instructions and result in higher forwarding rate. We believe such tradeoff is worth, because the control store still has enough space (75 %) to add other instructions for implementing

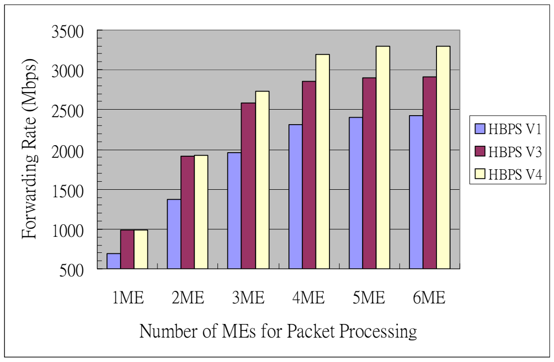

Figure 10. The speed-up of forwarding rate by adopts more MEs for packet processing.

other packet processing schemes.

- Forwarding rate with more ME's

Figure 10 compares the forwarding rates of *HBPS-V1, HBPS-V3,* and *HBPS-V4* while more ME's are adopted for packet processing. The optimization techniques make *HBPS-V3* and *HBPS-V4* always outperforms *HBPS-V1*. Table I shows the average length of SRAM Read Bus FIFO. The average length of *HBPS-V4* is less than *HBPS-V3* when more than 3MEs are used. And *HBPS-V4* also outperforms *HBPS-V3* in those cases. The technique that distributes the memory pressure to other memory interface mainly works in the case that average length of the SRAM command bus is relatively long.

- Limitation of the HBPS

The bottleneck of HBPS is the memory accesses required for processing packets. However, it can be solved by hardcoding more data structure into control store because only 25 % of the control store is needed in the original un-optimized HBPS. Thus, how to hardcode more data structure into the remaining 75% control store is left as our future work.

VII. Conclusion

Network processor is a promising solution to develop high speed routers. While developing packet processing schemes on network processors, many programming issues need to be considered. Implementations without any optimization technique can not achieve their ideal performance. In this paper, we surveyed and applied several programming techniques to enhance the performance of HBPS. By considering these programming issues, we can increase the forwarding rate of HBPS by 42 %. Although these techniques are evaluated by using the packet classification problem, they also work for other packet processing tasks.

References

[1] Y.-K. Chang, "Efficient Multidimensional Packet Classification with Fast Updates," *IEEE Transactions on Computers*, Vol. 58, No. 4, pp. 463-479, April 2009.
[2] Intel Corporation, "Intel® IXP2400 Network Processor Hardware Reference Manual," November 2003.
[3] Intel Corporation, "Intel® IXP2400/IXP2800 Network Processors Development Tools User's Guide", March 2004.
[4] Intel Corporation, "Intel® IXP2400/IXP2800 Network Processors Microengine C Language Support Reference Manual," November 2003.
[5] D. Liu, Z. Chen, B. Hua, N. Yu, X. Tang, "High-performance Packet Classification Algorithm for Multithreaded IXP Network Processor," *ACM Transactions on Embedded Computing Systems*, Vol. 7, Issue 2, Article 16, February 2008.
[6] RadiSys Corporation, "ENP-2611 Hardware Reference", August 2003.
[7] Z. X. Tan, C. Lin, H. Yin *et al.*, "Optimization and benchmark of cryptographic algorithms on network processors," *IEEE Micro,* vol. 24, no. 5, pp. 55-69, Sep-Oct, 2004.
[8] D. E. Taylor, "Survey and Taxonomy of Packet Classification Techniques," *ACM Computing Surveys*, Volume 37, Issue 3, pp. 238-275, September 2005.
[9] D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," *IEEE/ACM Transactions on Networking*, Vol. 15, Issue 3, pp. 499-511, June 2007.
[10] http://www.baymicrosystems.com/
[11] http://www.lsi.com/networking_home/networking_products/network_processors/index.html
[12] http://www.netronome.com/pages/network-flow-processors
[13] http://www.xelerated.com/en/hx/