High-performance Packet Classification Algorithm for Many-core and Multithreaded Network Processor

Duo Liu, Bei Hua, Xianghui Hu, and Xinan Tang†

Department of Computer Science and Technology
University of Science and Technology of China
Hefei, China, 230027
{liuduo, xhhu} @mail.ustc.edu.cn
bhua@ustc.edu.cn

† Intel Compiler Lab
SC12, 3600 Juliette Lane
Santa Clara, California, 95054
xinan.tang@intel.com

ABSTRACT

Packet classification is crucial for the Internet to provide more value-added services and guaranteed quality of service. Besides hardware-based solutions, many software-based classification algorithms have been proposed. However, classifying at 10Gbps speed or higher is a challenging problem and it is still one of the performance bottlenecks in core routers. In general, classification algorithms face the same challenge of balancing between high classification speed and low memory requirements. This paper proposes a modified Recursive Flow Classification (RFC) algorithm, **Bitmap-RFC**, which significantly reduces the memory requirements of RFC by applying a bitmap compression technique. To speed up classifying speed, we experiment on exploiting the architectural features of a many-core and multithreaded architecture from algorithm design to algorithm implementation. As a result, Bitmap-RFC strikes a good balance between speed and space. It can not only keep high classification speed but also reduce memory space significantly.

This paper investigates the main NPU software design aspects that have dramatic performance impacts on any NPU-based implementations: *memory space reduction*, *instruction selection*, *data allocation*, *task partitioning*, and *latency hiding*. We experiment with an architecture-aware design principle to guarantee the high performance of the classification algorithm on an NPU implementation. The experimental results show that the Bitmap-RFC algorithm achieves 10Gbps speed or higher and has a good scalability on Intel IXP2800 NPU.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; C.2.6 **[Internetworking]**: Routers: Packet Classification; D.1.3 [**Programming Languages**]: Concurrent Programming – *parallel programming*; D.2.2 [**Software Engineering**]: Design Tools and Techniques;

CASES'06, October 23-25, 2006, Seoul, Korea.

General Terms

Performance, Algorithms, Design, Experimentation

Keywords

Network processor, packet classification, architecture, multithreading, thread-level parallelism, embedded system design

1. INTRODUCTION

Nowadays the ever-increasing demand for quality of service (QoS) and network security, such as policy-based routing, firewall, and virtual private network (VPN), edge and core routers are required first to classify packets into flows according to a classifier and then to process them differently. As the new demand for supporting triple play (voice, video, and data) services arises, the pressure for the routers to perform fast packet classification becomes higher and higher. However, it is still challenging to perform packet classification at 10Gbps speed or higher by an algorithmic approach, whereas hardware-based solutions are both expensive and inflexible.

As the network processor unit (NPU) emerges as a promising candidate for a networking building block, NPU opens a new venture to explore thread-level parallelism to attack the performance bottleneck of classification. NPU is expected to retain the same high performance as that of ASIC and to gain the time-to-market advantage from the programmable architecture. Many companies, including Intel[15], Freescale[10], AMCC[3] and Agere[1] have developed their own programmable NPUs. Even though the NPU vendors only achieved limited success in terms of the market value, the NPU based technology has been widely used in commercial routers[4][7][14]. Therefore, an NPU based classification algorithm is worth of further study to realize NPU potential, which promises to provide a total solution for packet processing including forwarding and classification.

In general, there are four types of packet classification algorithms: grid-of-tries[22], bit vector linear search[6][18], cross-producting[22] and recursive flow classification (RFC)[12], and decision tree approaches[11][20]. All these algorithms focus on striking balance between space and speed to achieve optimal algorithmic performance. However, little work has been done in parallelizing these algorithms on many-core and multithreaded NPU architectures. Furthermore, the previous performance results

collected on a general-purpose CPU cannot be directly applied to the parallel architectures, especially on the many-core and multithreaded NPU architecture. New efforts are therefore required to design parallel packet classification algorithms for the many-core and multithreaded architecture, which normally provides hardware-assisted multithreading support to execute thread-level parallelism for hiding the memory-access latency.

In this paper, we propose an architecture-aware classification algorithm that exploits the NPU architectural features to reduce the memory-access times as well as hide the memory-access latency. Particularly, we adopt a system approach in designing such an efficient classification algorithm for the Intel IXP2800. We use the interdisciplinary thinking to find the best solution in each algorithm decision point, from algorithm design to algorithm implementation.

A good classification algorithm for an NPU must at least take into account the following interdisciplinary aspects: *classification characteristics*, *parallel algorithm design*, *multithreaded architecture*, and *compiler optimizations*. We believe that high performance can only be achieved through close interaction among these interdisciplinary factors. For example, RFC[12] is so far the fastest packet classification algorithm in terms of the worst-case memory access times. Its table read operations form a reduction tree (root at the bottom), where the matching of a rule involves walking through the tree from the root to the leaves. The search of the tree is easy to be parallelized, because

1) nodes on the same level can potentially run in parallel;

2) nodes on different levels can also run in parallel.

On a multithreaded architecture, latency hiding can be utilized in three ways [23][24][25]. First, two *parallel* memory accesses can be issued consecutively, and thus the latency of the first memory access can be partially hidden by that of the second memory access. Second, the latency of one memory access of a thread can be overlapped with another thread's execution. Third, execution of ALU instructions can be overlapped with time spent on other outstanding memory accesses.

By taking advantage of the tree characteristics inherited in the RFC algorithm and the latency hiding ability of multithreaded architecture, multiple read operations can be issued simultaneously from a single thread, and the read operations from different threads can also be issued to overlap their execution. In doing so, the long latencies caused by multiple memory accesses can be partially hidden, thus the RFC algorithm is an eligible candidate for an NPU based parallel implementation.

Even though the advent of many-core and multithreaded NPU has given rise to a new paradigm for parallel algorithm design and implementation, the results of general-purpose multi-processing research are not directly applicable to such system-on-chip (SOC) based many-core and multithreaded architectures due to their specific processing requirements [2][17]. This potential of great performance improvement motivates the development of an architecture-aware classification algorithm that exploits the unique architectural properties of an NPU to achieve high performance. **Bitmap-RFC** is such an NPU-aware IPv4 packet classification algorithm specifically designed to exploit the architectural features of the SOC based many-core and multithreaded systems.

Because an NPU is an embedded SOC with modest memory space, reducing memory footprint is the highest priority for almost every networking application. Furthermore, saving memory space opens other optimization opportunities for reducing the memory-access latency. For example, moving data from DRAM to SRAM on the Intel IXP2800 can save about 150 cycles for each memory access. Considering that the RFC algorithm requires explosive memory space when the number of classification rules becomes large, we introduce bitmap compression[8] to the RFC algorithm to reduce its table size so that the Bitmap-RFC can take advantage of faster SRAM for achieving high performance.

In the Bitmap-RFC implementation, we carefully investigate the following optimization opportunities that are directly related to any NPU-based network algorithm implementations: *space reduction*, *instruction selection*, *data allocation*, *task partitioning*, and *latency hiding*. For each opportunity, we explore the specific design space that might have trouble spots in Bitmap-RFC implementation. After evaluating these design decisions, we come up with a highly efficient time-space balanced packet classification algorithm, Bitmap-RFC, which is designed to run efficiently on the Intel IXP2800. The high-performance of the resulting algorithm is achieved through a process of design space exploration by considering application characteristics, efficient mapping from the algorithm to the target architecture, and applying source code transformations with both manual and compiler optimizations.

To summarize, the goal of this paper is to design and implement a high-performance packet classification algorithm on a many-core and multithreaded NPU through the system approach. We identify the key design issues in implementing such an algorithm and exploit the architectural features to address these issues effectively. Although we experiment on the Intel IXP2800, the same high-performance can be achieved on other similar NPU architectures [1][3][10]. The main contributions of the paper are as follows:

- A scalable packet classification algorithm is proposed and efficiently implemented on the IXP2800. Experiments show that its speedup is almost linear and it can run even faster than 10Gbps.
- Algorithm design, implementation, and performance issues are carefully studied and analyzed. We apply the systematical approach to address these issues by incorporating architecture awareness into parallel algorithm design.

To the best of our knowledge, Bitmap-RFC is the first packet classification implementation that achieves 10Gbps speed on the Intel IXP2800 for a classifier as large as 12,000 rules. Our experiences may be applicable to parallelizing other networking applications on other many-core and multithreaded NPUs as well.

The rest of this paper is organized as follows. Section 2 introduces related work on algorithmic classification schemes from the NPU implementation point of view. Section 3 formulates the packet classification problem and briefly introduces the basic ideas of the RFC algorithm. Section 4 presents the Bitmap-RFC algorithm and its design space. Section 5 discusses design decisions made related to NPU-based Bitmap-RFC implementation. Section 6 gives simulation results and

performance analysis of Bitmap-RFC on the Intel IXP2800. Section 7 presents guidance on effective network application programming on NPU. Finally, section 8 concludes and discusses our future work.

2. RELATED WORK

Prior work on classification algorithms have been reported in [5][6][11][12][18][20][22][27]. Below we mainly compare *algorithmic* classification schemes, especially from the NPU implementation point of view.

Trie-based algorithms, such as grid-of-tries[22], build hierarchical radix tree structures where if a match is found in one dimension another search is started on a separate tree pointing to another trie. In general, trie-based schemes work well for single-dimensional searches. However, their memory requirements increase significantly with the increase in the number of search dimensions.

Bit vector linear search algorithms[6][18] treat classification problem as an n-dimensional matching problem and search each dimension separately. When a match is found in a dimension, a bit vector is returned identifying the match and the logical AND of the bit vectors returned from all dimensions identifies the matching rules. However, fetching the bit vectors requires wide memory and wide buses, and thus are memory intensive. This technique is more profitable for ASIC than for NPU because the NPU normally has limited memory and bus width.

Hierarchical Intelligent Cuttings (HiCuts)[11] recursively chooses and cuts one searching dimension into smaller spaces, and then calculates the rules that intersect with each smaller space to build a decision tree that guides the classifying process. HyperCuts[20] improves upon HiCuts, in which each node represents a decision point in the multi-dimensional hypercube. HyperCuts attempts to minimize the depth of the decision tree by extending the single-dimensional search into a multi-dimensional one. On average HiCuts and HyperCuts achieve good balance between speed and space, however they require more memory accesses than RFC in the worst case.

RFC algorithm[12], which is a generalization of cross-producting[22], is so far the fastest classification algorithm in terms of the worst-case performance. Because the worst-case performance is used as one of the most important performance metrics of network systems[19], we base our classification algorithm on RFC to guarantee the worst-case performance, and then apply bitmap compression to reduce its memory requirement to conquer the problem of memory explosion.

Bitmap compression has been used in IPv4 forwarding[8][9] and IPv6 forwarding[13]. Recently it is applied to classification[21], which is the closest in spirit to ours in that all use bitmaps to compress redundant storage in data structure. However, previous methods cannot solve the performance bottleneck caused by searching the compressed tables, and thus additional techniques have to be introduced to address the inefficiency of calculating the number of bits set in a bitmap. For example, the Lulea[8] algorithm introduces a summary array to pre-compute the number of bits set in the bitmap, and thus it needs an extra memory access per trie-node to search the compressed table. The Bitmap-RFC employs a built-in bit-manipulation instruction to calculate the number of bits set at runtime, and thus is much more efficient than Lulea's in terms of time and space complexity.

3. PROBLEM STATEMENT

Packet classification is the process of assigning a packet to a flow by matching certain fields in the packet header with a classifier. A classifier is a database of N rules, each of which, R_j, $j=1, 2, \ldots, N$, has d fields and an associated action that must be taken once the rule is matched. The i^{th} field of rule R_j, referred to as $R_j[i]$, is a regular expression pertaining to the i^{th} field of the packet header. The expression could be an exact value, a prefix, or a range. A packet P is said to match a rule R_j if each of the d fields in P matches its corresponding field in R_j. Since a packet may match more than one rule, a priority must be used to break the ties. Therefore, packet classification is to find a matching rule with the highest priority for each incoming packet.

Table 1. Example of a simple classifier

Rule#	F_1	F_2	F_3	***Action***
R_1	001	010	011	Permit
R_2	001	100	011	Deny
R_3	01*	100	***	Permit
R_4	***	***	***	Permit

Let S represent the length of a bit string concatenated by the d fields of a packet header, then the value of this string falls into [0, 2^S-1]. Searching a particular rule based on directly indexing on the string of concatenated fields (CF-string for short hereinafter) is out of the question when S is big. The main idea of RFC is to split such a one-time mapping into multi-phase mapping in order to reduce a bigger search space into multiple smaller ones. Each mapping phase is called a *reduction*, and the data structure formed by multi-phase mapping is called a *reduction tree*. After multi-phase mapping, S-bit CF-string is mapped to a T-bit ($T << S$) space.

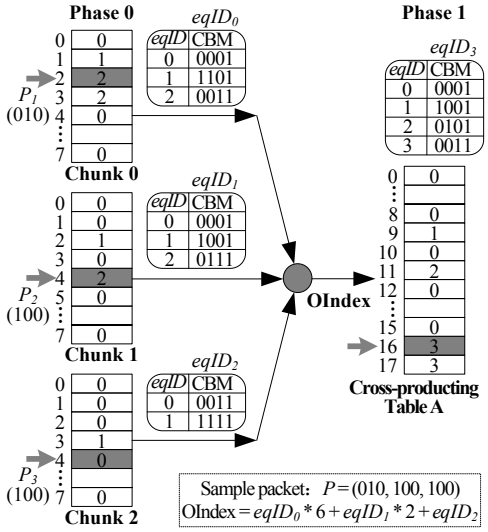

Figure 1. A two-phase RFC reduction tree

Let us use a simple example to illustrate the building process of a reduction tree. Figure 1 is a two-phase RFC reduction tree constructed from the classifier defined in Table 1, in which each

rule has three fields and each field is 3 bits long. The reduction tree is formed by two phases:

In the first phase (Phase 0), each field (F_1-F_3) is expanded into a separate preprocessed table (Chunk 0-2). Each chunk has an accompanying equivalence class ID (*eqID*) array, and each chunk entry is an index to its *eqID* array (table). Each entry of $eqID_i$ is a bit vector (**Class Bitmap, CBM**) recording all the rules matched as if the corresponding index to the *Chunk* array is used as input. For example, the value of the first entry of Chunk 0 is 0, which points to the first element of array $eqID_0$ whose bitmap is '**0001**'. Each bit in a bitmap corresponds to a rule, with the most significant bit corresponding to R_1, and the least significant bit to R_4. Each bit records whether the corresponding rule matches or not for a given input. Thus, bitmap '**0001**' means only rule R_4 matches when index 0 of Chunk 0 is used as F_1 input. Similarly, the first entry of Chunk 2 has value 0, and it points to the first entry of $eqID_2$ whose bitmap is '**0011**', indicating only rules R_3 and R_4 match if index 0 of Chunk 2 is used as input for field F_3.

In the second phase (Phase 1), a ***cross-producting table*** (CPT) and its accompanying *eqID* table are constructed from the *eqID* tables built in Phase 0. Each CPT entry is also an index, pointing to the final *eqID* table whose entry records all the rules matched when the corresponding index is concatenated from "$eqID_0eqID_1eqID_2$". For instance, the index of the first entry of CPT is 0, calculated from concatenating three bit strings '00'+'00'+'00'. The rules matched can be computed as the intersection of $eqID_0$ [0]('0001'), $eqID_1$ [0]('0001'), and $eqID_2$[0]('0011'). The result is '**0001**', indicating rule R_4 matches when `000-000-000' is used as input for the three fields F_1, F_2, and F_3.

The lookup process for the sample packet *P* (010,100,100) in Figure 1 is as follows:

1) use each filed, P_1, P_2 and P_3 (i.e., 010,100,100) to look up Chunk 0-2 to compute the index of cross-producting table ***A*** by $\mathbf{Chunk_0[2]*3*2+Chunk_1[4]*2+Chunk_2[4]}$, which is 16;

2) the value of CPT[16] is 3 and it is used as an index to $eqID_3$. The result of '0011' indicates that rules R_3 and R_4 match the input packet ***P***. Finally, R_3 is returned as it has higher priority than R_4 according to the longest match principle.

Note that *eqID* tables built in Phase 0 are only used to build the CPT A and will not be used thereafter. Careful readers may notice that there are many repetitions in table CPT A. For example, the first nine entries of CPT A have the same value 0. These redundant data may occupy a lot of memory space when the number of rules increases. We will use ***Independent Element*** (IE) to denote distinct *eqID* index in CPT tables. The most natural way to reduce the data redundancy is to store a sequence of consecutively identical entries as one and use other auxiliary information to indicate the start and end of the sequence.

Normally the compression of the table is usually at the cost of increased table searching time. Additional techniques must be used to solve this problem. Moreover, architectural features of NPU must be exploited to further optimize the algorithm performance. Therefore, we state our classification problem as follows:

1) apply a compression technique to RFC's cross-producting tables to reduce the data redundancies;

2) exploit the NPU architectural features to achieve high classification speed, especially at 10Gbps speed or higher on Intel IXP 2800.

4. BITMAP-RFC ALGORITHM

4.1 Algorithm Description

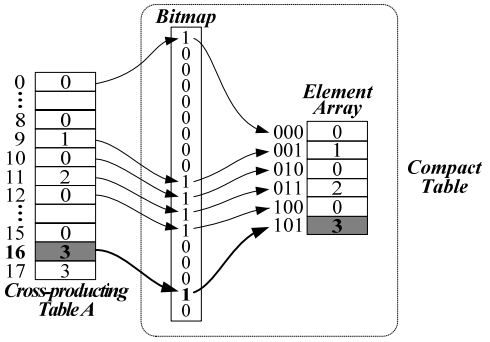

Figure 2. Illustration of Bitmap-RFC algorithm

Figure 2 illustrates the basic idea of ***Bitmap-RFC*** algorithm. A bit vector called ***Bitmap*** is used to track the appearance of independent elements (IEs) in CPT. A sequence of consecutively identical elements are compressed and stored as one element in an array called ***Element Array***. The data structure consisting of ***Bitmap*** and ***Element Array*** is called ***Compact Table***.

Each bit of Bitmap corresponds to an entry in CPT, with the least significant bit (LSB) corresponding to the first entry. Bitmap is formed starting from the LSB: a bit is set to '1' if its corresponding entry in CPT has an IE different from its previous one. Therefore, a bit set in Bitmap indicates that a different sequence of consecutively identical elements starts at the corresponding position in CPT. Whenever a bit is set, its corresponding IE is added in Element Array. '010203' is the resultant Element Array for CPT A listed in Figure 2.

Since the length of Bitmap increases with the size of CPT, and scanning longer bit vector (string) introduces higher overhead, we divide a CPT into segmented sub-CPTs with a fixed size, and employ the bitmap compression technique to compress each sub-CPT into an entry of Compact Table.

4.2 Data Structure

Figure 3 shows the data structure used in Bitmap-RFC, which comprises ***Compact Table*** and ***Accessory Table***. We will use ***Compressed CPT*** (CCPT) to denote this data structure in the rest of this paper. Only one entry of Compact Table is illustrated in this figure due to space limit. Accessory Table is only needed when Element Array is full. In that case, the last two units of Element Array are used as an address field pointing to the Accessory Table. ***Second Memory Accesses Ratio*** (SMAR) is defined as the ratio of the number of elements in Accessory Tables to the total number of elements in both tables.

The sizes of sub-CPT and CCPT entry are two important design parameters, which have great impacts on SMAR of the algorithm, and also have close relation to both the set of rules and

the logical width of memory. Therefore, these two parameters must be properly chosen before implementing Bitmap-RFC on a particular platform.

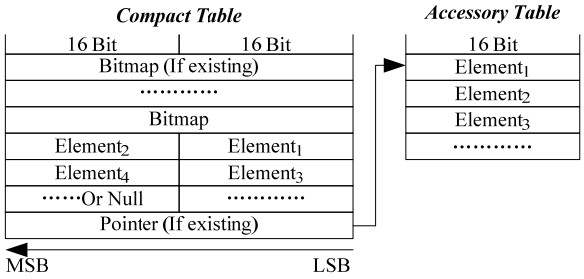

Figure 3. Data structure for Bitmap-RFC algorithm

The reason for putting Bitmap and Element Array together is to obtain efficient memory access by utilizing a useful feature of NPU such as the IXP2800. On the IXP2800, adjacent SRAM locations up to 64 bytes can be fetched in one SRAM read instruction, so memory accesses can be reduced by putting them together so that they can be fetched by one SRAM access.

In order to read the Compact Table efficiently from the memory, the entry size of Compact Table should be a multiple of the logical width of memory, which is 32 bits for SRAM and 64 bits for DRAM on the IXP2800. Thus in the case of IXP2800, the size of sub-CPT could be 32, 64, 96, 128 bits and so on.

4.3 Bitmap-RFC Lookup

Figure 4 gives the pseudo code of Bitmap-RFC search algorithm. Since the searching process of phase 0 in Bitmap-RFC is similar to that in RFC, we only discuss the search algorithm for CCPT. Some constants are predefined to facilitate understanding of the algorithm. We predefine the size of Bitmap as one word (32 bits), the size of Element Array as four words, containing maximally 8 elements of 16-bits each. The entry size of Compact Table is five words, which equals to the sum of Bitmap's and Element's size. That means one memory access should fetch 5 words from SRAM into the Compact Table (line 3), and we borrow an array notation, CompactTable[0..4] to denote the Bitmap and Element Array defined in Figure 3, i.e. CompactTable[0] stores Bitmap (line 6) and from CompactTable[1] to CompactTable[4] store the Element Array with 8 entries (line 8).

First, **Oindex** is divided by the size of sub-CPT named (*SubLen)* in line 2 to compute the index (**Cindex**) of CCPT (because each sub-CPT is compressed into one entry in CCPT). Second, **Cindex** is used to look up the CCPT and get the data (Bitmap and Element Array) from a Compact CPT Table entry (line 3). If the number of bit set in a Bitmap is 1, which is computed by function POP_COUNT, then there is only one IE in the Element Array, i.e. Element[0] (lines 9-10) is the final searching result. Otherwise, **BitPos**, the position of the bit corresponding to the searched entry in sub-CPT, is calculated by **Oindex** (line 4). Then the number of bits set (**PositionNum**) from bit 0 to bit **BitPos** is calculated by an intrinsic function POP_COUNT (line 12). **PositionNum** is used as an index to look up the Element Array (lines 13-18). If an IE being searched is in the Element Array (no greater than 8 entries as shown in line 13), the result is returned from line 14; otherwise, the Accessory Table needs to be searched (lines 16-19). It is worth mentioning that each word contains two Element Array elements because the size of each element is 16 bits.

Let's take the example given in Figure 2 to illustrate the searching process, where the size of CPT A is 18. Since each compact table element can hold as many as 32 CPT entries, one sub_CPT is enough for such a case. Because only one entry exists in the Compact Table, **Cindex** is zero and **BitPos** and **Oindex** are the same. For the input packet ***P***, **BitPos** equals to 16. The **PositionNum** is calculated by counting the number of bits set from bit 0 to bit 16 (line 12) for the Bitmap listed in Figure 2.

If the element array is organized as shown in Figure 2, in which *six* IEs are stored in the Element array, then line 14 will be executed. On the other hand, if each element array stores less than *six* entries, lines 16-19 would be executed, and the Accessory Table would be visited to locate the corresponding IE. Therefore, the size of the compact table has dramatic impacts on the number of memory accesses occurred during the search.

```
Bitmap-RFC_CCPT_Search (IN Oindex,OUT IE) {
1:    Current_Node = CCPT CompactTable;
2:    Cindex = Oindex / SubLen; // the index to Compact Table
3:    CompactTable[0..4] = Read_CCPT(Current_Node,Cindex);
4:    BitPos = GitBitPos(Oindex);
5:    //! allocate an array Bitmap[1] to store Bitmap
6:    Bitmap[0] = CompactTable[0];
7:    //! allocate an array Element[8] to store Element Array
8:    Element[0..7] = CompactTable[1..4];
9:    if (POP_COUNT(Bitmap[0]) == 1)
10:       return Element[0];
11:   else {
12:       PositionNum = POP_COUNT(Bitmap[0],BitPos) - 1;
13:       if ( PositionNum < 8) {
14:          return Element[PositionNum];
15:       }
16:       else { // IE being searched is in the AccessoryTable
17:          Current_Node   =CCPT AccessoryTable;
18:          AccessoryIndex=GetAccessoryIndex(PositionNum);
19:          return Read_CCPT(Current_Node,AccessoryIndex);
20:       }
21:   }
} // Bitmap-RFC_CCPT _Search
```

Figure 4. Pseudo code for Bitmap-RFC search operation

5. NPU-AWARE DESIGN AND IMPLEMENTATION

Figure 5 draws the components of the Intel IXP2800[15], in which 16 Microengines (MEs), 4 SRAM controllers, 3 DRAM controllers, and high-speed bus interfaces are shown. Each ME has eight hardware-assisted threads of execution, and 640-words local memory of single-cycle access. There is no cache on each ME. Each ME uses the shared buses to access off-chip SRAM and DRAM. The average access latency for SRAM is about 150 cycles, and that for DRAM is about 300 cycles. We implemented Bitmap-RFC algorithm in the MicroengineC language, which is a subset of the ANSI C plus parallel and synchronization extensions, and simulated it on a cycle-accurate simulator. In the following, we will discuss the crucial issues that have great

impacts on algorithm performance when we implement the Bitmap-RFC on the IXP2800 NPU.

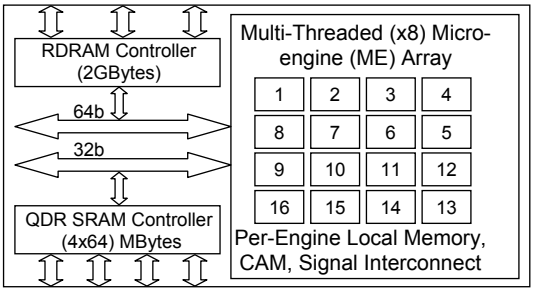

Figure 5. IXP2800 component diagram without I/O interfaces

5.1 Memory Space Reduction

The NPU is generally an embedded SOC whose memory size is limited. SRAM and DRAM are two types of commonly used NPU memory, whose size is of megabyte magnitude, and can be used to store classification tables. On the Intel IXP2800, the size of DRAM is approximately eight times as large as that of SRAM, however its latency is approximately twice as long as that of SRAM. Therefore, the memory access latency can be greatly reduced if the reduction tree is stored in SRAM. The bitmap compression is an enabling technique to make it happen.

From the discussion in Section 4.2 and 4.3, the performance of Bitmap-RFC is greatly affected by the data structure design in CCPT, especially the size of sub-CPT. We choose the size of bitmap as 32 bits to match the instruction word width of the NPU ISA. The size of the *Element Array* is 4 words containing at most 8 elements, because the number of bits set in a sub-CPT is most likely no greater than 8. This is an observation we have noticed for the classification rules we have experimented on (see Figure 7). Please note that the size of the *Element Array* affects the second memory access ratio (SMAR) and the effectiveness of memory compression. The larger the *Element Array* is, the less the SMAR is, and the more memory space is required to store CCPT. The size of *Element Array* should be adjustable based on the distribution of the number of bits set in the bitmap.

5.2 Instruction Selection

Searching CCPT requires computing **PositionNum**. It is a time-consuming task in traditional RISC/CISC architecture, as it usually takes more than 100 RISC/CISC instructions (ADD, SHIFT, AND, and BRANCH) to compute the number of bits set in a 32-bit register. Without direct hardware support, calculation of **PositionNum** will become a new performance bottleneck in Bitmap-RFC algorithm.

Fortunately, there is a powerful bit manipulation instruction in IXP2800 called **POP_COUNT**, which can calculate the number of bit set in a 32-bit register in 3 cycles. With **POP_COUNT**, the number of instructions used to compute **PositionNum** is reduced by more than 97% compared with other RISC/CISC implementations. This is essential for the Bitmap-RFC algorithm to achieve the line rate.

NPUs that do not have **POP_COUNT** normally have another bit-manipulation instruction, **FFS,** which can find the first bit set in a 32-bit register in one clock cycle. With **FFS**, **PositionNum** can be calculated by looping through the 32 bits and continuously looking for the next first bit set.

Compared with RISC architecture, the NPU normally has much faster bit-manipulation instructions. Appropriately selecting these instructions can dramatically improve the performance of NPU-aware algorithms. However, current compiler technology cannot always generate such instructions automatically. It is the programmer's responsibility to select them manually through intrinsic or in-line assembly.

5.3 Data Allocation

The Intel IXP2800, like other NPUs, has a complex memory hierarchy that comprises, in the increasing order of memory access latency, single-cycle *local memory*, *scratchpad*, *SRAM*, and *DRAM*. For Bitmap-RFC implementation, the choice of using SRAM or DRAM, and where and how to distribute the compressed tables greatly affect the ultimate classification speed.

In addition to the aforementioned size and speed differences between SRAM and DRAM, different access granularities and memory alignment must be taken into account as well. For example, on the Intel IXP2800, SRAM is optimized and aligned for 4-byte word access, while DRAM is optimized for at least 16-byte burst access. Therefore, data structures must be optimized for the specific type of memory.

There are four independent SRAM controllers on the IXP2800 that allow parallel access, and three DRAM controllers, with each DRAM controller having four memory banks that can be accessed in an interleaved manner. To evaluate the performance impacts of parallel SRAM access and interleaved DRAM access, we designed the following six settings. Experimental results show that the fourth setting can meet the OC-192 speed even in the worst case.

- All the tables are stored in one SRAM controller;
- Tables are properly distributed on two SRAM controllers;
- Tables are properly distributed on three SRAM controllers;
- Tables are properly distributed on four SRAM controllers;
- All the tables are distributed on DRAM, and data structures are redesigned to facilitate the burst access;
- Tables are properly distributed on SRAM and DRAM in a hybrid manner.

Table 2. Data allocation scheme for the fourth setting

SRAM controller 0	Phase 0 Chunks & All Accessory Tables
SRAM controller 1	Compact Table of CCPT X'
SRAM controller 2	Compact Table of CCPT Y'
SRAM controller 3	Compact Table of CCPT Z'

We experimented on various data allocation schemes for the fourth setting, however, only the one shown in Table 2 is the best solution to achieve OC-192 speed. In this setting, three Compact Cross-producting Tables, CCPT_X', CCPT_Y', CCPT_Z', used in the second, and the third levels of the reduction tree, are distributed into three different SRAM controllers.

As discussed above, there is another useful feature that can be effectively exploited on the IXP2800: adjacent SRAM locations can be fetched in one SRAM read instruction (maximally 64 bytes). By designing the Compact Table size being 20 bytes (less than 64 bytes), memory vectorization optimization can be applied to significantly reduce the number of SRAM accesses.

5.4 Task Partitioning

There are two general ways to partition tasks onto multiple MEs on the Intel IXP2800: *multi-processing* and *context-pipelining*. Multi-processing involves two parallelizing techniques. First, multi-threading is applied to a task allocated to one ME. In an Intel IXP2800, a maximum of 8 threads can be used per ME. Secondly, a task can use multiple MEs if needed. For example, if a task needs 2 MEs, a maximum of 16 task threads can run in parallel. Each thread instance runs independently, assuming no other thread instances exist. Such a *run-to-completion* programming model is similar to the sequential one, and it is easy to be implemented. In addition, the workloads are easier to be balanced. However, threads allocated on the same ME must compete for shared resources, including registers, local memory, and command (data) buses. For example, if a task requires more local memory than one ME can support, the context-pipelining approach must be used instead.

Context-pipelining is a technique that divides a task into a series of smaller sub-tasks (contexts), and then it allocates them onto different MEs. These contexts form a linear pipeline, similar to an ASIC pipeline implementation. The advantage of context-pipelining is to allow a context to access more ME resources. However, the increased resources are achieved at the cost of communication between neighboring MEs. Furthermore, it is hard to perform such partitioning if workloads cannot be determined at compile time. The choice of which method to use should depend on whether the resources can be effectively utilized on all MEs.

The workloads of different phases in Bitmap-RFC are unbalanced, and thus the multi-processing scheme may achieve higher performance than context-pipelining. The simulation results shown in section 6.6 confirm this prediction.

5.5 Latency Hiding

Hiding memory latency is another key to achieving high-performance of Bitmap-RFC implementation. We hide the memory-access latency by overlapping the memory access with the ALU instructions calculating the **BitPos** in Bitmap in the same thread as well as memory access issued from other threads.

For instance, in Figure 4 operations listed in line 3 and line 4 can run in parallel so that the **BitPos** computation is hidden completely by the memory operation Read_CCPT(). Compiler based thread scheduling should be able to perform such an optimization automatically [23].

6. SIMULATION AND PERFORMANCE ANALYSIS

Due to privacy and commercial secrets, it is hard for us to access sufficient real classifiers. Luckily, the characteristics of real classifiers have been analyzed in [5][16][26][12]. Therefore, we could construct the synthetic ones for the core router accordingly [20]. In order to measure the performance impact of the Intel IXP2800 on the Bitmap-RFC algorithm, we experiment with the following implementations discussed from section 6.2 to 6.7.

6.1 Experimental Setup

The RFC algorithm needs six chunks (corresponding to 16 bits lower/higher src/dst IP address, 16 bits src/dst port number respectively) in Phase 0, two CPTs (CPT X and CPT Y) in Phase 1, and one CPT (CPT Z) in Phase 2. To compare with the RFC algorithm, we implemented 4-deminsional Bitmap-RFC packet classification with the same three phases. The compressed tables used by Bitmap-RFC are called CCPT X'/Y'/Z'.

Since the searching time of the RFC algorithm depends on the structure of the reduction tree, the number of memory access times is fixed for the RFC algorithm (9 times). As a result, more rules only affect the memory space needed. Therefore, three classifiers were constructed for the experiments presented in section 6.3 to 6.7. They are CL#1, CL#2, CL#3, and each has 1000, 2000, 3000 rules respectively.

Each sub-CPT has 32 entries for all experiments. Thus, the length of Bitmap is 32 bits. We allocate four long words (4*32 bits) for Element Array (eight Element units), so the size of Compact Table is 5 long words.

We use the minimal packets as the worst-case input [19]. For the OC-192 core routers a minimal packet has 49 bytes (9-byte PPP header + 20-byte IPv4 header + 20-byte TCP header). Thus, a classifying rate of 25.5Mpps (Million Packets per Second) is required to achieve the OC-192 line rate.

6.2 Memory Requirement of RFC and Bitmap-RFC

To find out the maximum number of rules that RFC and Bitmap-RFC can hold on an IXP2800, we use two metrics:

- The minimum number of rules that causes one of tables lager than 64MBytes (the size of one SRAM controller).
- The minimum number of rules that causes the total size of all tables larger than 256MBytes (the total size of SRAM).

The results for RFC and Bitmap-RFC are shown in Table 3.

Table 3. Comparison of memory requirements

Num. of Rules	Memory Requirement of CPT and CCPT (MB)						Total Memory Requirement (MB)	
	X	X'	Y	Y'	Z	Z'	RFC	Bitmap-RFC
5,700	59.2	18.5	25.8	8.1	***64.1***	16.0	149.9	43.4
8,050	80.7	25.2	64.3	20.1	127.7	39.9	***273.5***	86.0
12K	106.5	33.3	106.3	33.2	284.9	***71.2***	498.6	138.5
17K	191.0	59.7	185.0	57.8	570.7	178.4	947.6	***296.7***

When the number of rules exceeds 5,700, RFC cannot be implemented on IXP2800, since the memory requirement of table Z is bigger than 64MBytes. In addition, the total memory requirement of RFC will be bigger than 256MBytes when 8,050 rules are used.

When the number of rules exceeds 12,000, Bitmap-RFC cannot be implemented on IXP2800 when the size of sub-CPT is 32, since the memory requirement of table Z' is bigger than 64MBytes. In addition, the total memory requirement of Bitmap-RFC will be bigger than 256MBytes when 17,000 rules are used.

Since SRAM space required for Z' bank is over 64MBytes when 12,000 rules are used, the corresponding CPT must be split into two sub-tables appropriately to fit into two SRAM banks. Such a split is doable but at the cost of increasing the searching complexity. Therefore, the maximal number of rules should be less than 12,000 when the Bitmap-RFC algorithm is used in practice.

6.3 Relative Speedups

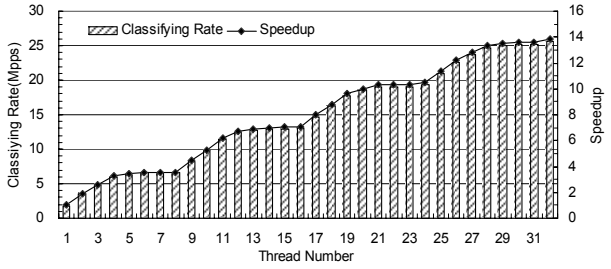

Figure 6. Bitmap-RFC classifying rates and relative speedups

Figure 6 shows the classifying rates and relative speedups of Bitmap-RFC using the minimal packet size on the Intel IXP2800 based on CL#1. The data were collected after all optimizations previously mentioned were applied and the RFC reduction tree is stored in four SRAM channels. The speedup is almost linear and classification speed reaches up to 25.65Mpps for 32 threads. The reason of sub-linear speedup is partially caused by the saturation of command request FIFOs and SRAM buses. The SRAM bus behavior is illustrated in Table 4, in which the FIFO fullness ratio can be used to indicate the degree of saturation.

Table 4. Bus behavior measured by fullness ratio

Num. of Threads	ME CMD Request FIFO Fullness Ratio (%)	SRAM Controller Read FIFO Fullness Ratio (%)
1	11.7	2.7
8	40.1	11.1
16	41.5	20.9
24	41.4	36.9
32	41.3	47.9

Bitmap-RFC is a memory-bound algorithm in which each thread issues multiple outstanding memory requests per packet. If these memory requests cannot be processed in time, the classification performance will drop. Taken CL#1 as an example, the ME CMD Request FIFO fullness ratio increases from 11.7% for one thread to 40.1% for eight threads. That is, in the 8-thread mode, a read memory request stays almost four times longer in the FIFO than it does in the 1-thread mode. Similarly, the SRAM Controller Read FIFO fullness ratio increases from 2.7% for one thread to 47.9% for 32-threads, that is, in 32-thread, a read memory request stays nearly 18 times longer in FIFO than it does in the 1-thread mode. This architectural constraint prevents the Bitmap-RFC algorithm from having a 100% linear speedup.

Because our implementation is well over the line-rate speed when 4 MEs (32 threads) are fully used, we want to know the exact minimum number of threads required to meet the OC-192 line rate. Table 5 shows the minimum number of threads required for the three classifiers. On average all of the classifiers need 20 threads to reach OC-192 line rate.

Table 5. Minimal threads required for supporting line rate

Classifier	Minimum Number of Threads	Classifying rate (Mpps)	
		Single thread	Multithreads
CL#1	20	1.97	26.65
CL#2	20	1.94	25.74
CL#3	20	1.95	25.77

Considering there are sixteen MEs on the Intel IXP2800, three MEs for IPv4 packet classification use only less than 1/5 of the entire ME budget. Therefore, Bitmap-RFC leaves enough room for other networking applications, such as packet forwarding and traffic management, to meet the line-rate performance.

6.4 Instruction Selection

Table 6. Classifying rates (Mpps) of POP_COUNT vs. FFS

		1 ME	2 MEs	4 MEs	8 MEs
CL#1	FFS	5.04	10.07	20.03	33.15
	POP_COUNT	6.54	12.85	25.65	33.35
	Improvement	30%	29%	28%	1%
CL#2	FFS	4.49	9.03	17.85	32.87
	POP_COUNT	6.38	12.77	25.09	33.33
	Improvement	42%	41%	41%	1%
CL#3	FFS	4.48	8.96	17.54	32.55
	POP_COUNT	6.38	12.77	25.09	33.33
	Improvement	43%	43%	43%	2%

Table 6 shows the classifying rates of the worst-case input packets by using two different instructions: **POP_COUNT** and **FFS** respectively. The testing is done for three rule sets. In general, the classifying rate of **POP_COUNT** based implementation is higher than that of **FFS** based implementation. On average, the performance improvement using **POP_COUNT** can be as high as 43% compared to FFS based implementation. The exception is on 8MEs, because the utilization rate of four SRAM controllers has reached 96% in both cases, and the SRAM bandwidth becomes a new performance bottleneck, which shadows other performance improvement factors. The same impact can also be observed in other performance tables reported in section 6.6, and 6.7.

The lower classifying rate of **FFS** is because computational time of **PositionNum** depends on the number of bits set in the Bitmap. The more bits are set, the more instructions are executed at runtime. This shows that an architecture-aware algorithm needs to consider the instruction selection to facilitate its implementation because those instructions might have a significant impact on the performance of the algorithm.

According to the three classifiers we experimented, we found out the number of bits set in Bitmap did not exceed three on average. The distribution of the number of bits set in Bitmap is provided in Figure 7 in which it shows that

- the majority of Bitmaps have only one bit set;
- most likely the number of bits set is less than 8.

Otherwise, the speed improvement of POP_COUNT vs. FFS should be even higher.

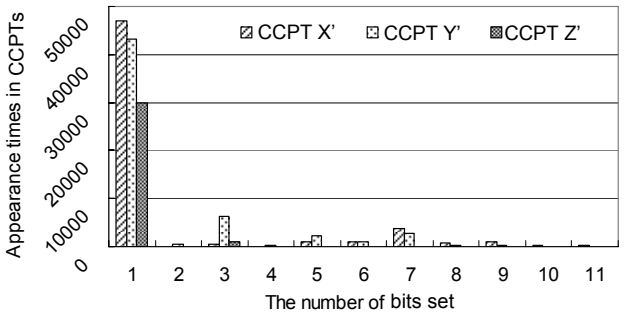

Figure 7. Distribution of the number of bits set in Bitmap

6.5 Memory Impacts

The simulation shows that our algorithm cannot support the OC-192 line rate in the worst case if DRAM alone is used. The culprit is the DRAM push bus, which is shared by all MEs for reading CCPT. Instead, we simulated the following six data allocation schemes using the worst-case minimal-packet input on the IXP2800. Table 7 shows the simulation results. We found out:

Table 7. Classifying rates (Mpps) on different data allocations

	1 ME	2 MEs	4 MEs	8 MEs
1-SRAM	6.49	9.65	9.63	9.576
2-SRAM	6.50	12.94	18.63	18.15
3-SRAM	6.50	12.99	20.40	20.10
4-SRAM	6.54	12.85	25.65	33.35
Hybrid-1	7.03	12.66	12.66	12.63
Hybrid-2	7.13	12.82	12.80	12.78

- The 1-SRAM/2-SRAM/3-SRAM table allocation schemes cannot support OC-192 line rate if the number of MEs is no greater than 4. Because the utilization rate of a single SRAM controller is up to 98% at 4 MEs and 8 MEs, the single SRAM bandwidth is the performance bottleneck of these schemes.
- Only the 4-SRAM configuration obtains almost linear speedup from 1 ME up to 8 MEs. Additionally, the utilization rates of four SRAM controllers are all approximately 96% when 8 MEs are used, indicating the potential speedup could be even greater if the system could have more SRAM controllers.
- We experimented two kinds of hybrid table allocations. The Hybrid-1 configuration stores the preprocessing tables (Chunks 0-5) in SRAM and CCPT X'/Y'/Z' in DRAM. The Hybrid-2 configuration stores Chunks 0-5 and CCPT X'/Y' in SRAM and CCPT Z' in DRAM. The simulation shows that both of them cannot support the OC-192 line rate in the worst case either.

6.6 Task Partitioning

The communication method in context-pipelining could be a scratch ring or a next-neighbor ring (FIFO). Two types of context-pipelining partitioning were implemented based on next-neighbor ring. We divided the whole classifying task into pieces according to (1) the algorithm logic; (2) the number of memory accesses required per ME. The partitioning configurations are as follows:

1) The first ME is for the search of all preprocessed tables in Phase 0 and the second is for search the CCPT X'/Y'/Z' in the following phases.
2) The first ME is for the search of half preprocessed tables, the second ME is for the search of rest preprocessing tables and CCPT X', and the third ME is for the search of CCPT Y'/Z'.

Because the task of Bitmap-RFC in any stage is not well-balanced, it is extremely difficult to partition the workload evenly. In addition, the communication FIFOs also add the overhead. Each ME must check whether the FIFO is full before a **put** operation and whether it is empty before a **get** operation. These checks take many clock cycles when context-pipelining stalls. Table 8 shows the simulation results using different task allocation policies.

Table 8. Classifying rates (Mpps) of multi-processing vs. context-pipelining

	2 MEs	3MEs	4 MEs	6 MEs
Multi-processing	12.85	19.32	25.65	33.73
Context-pipelining-1	12.31	--	--	--
Context-pipelining-2	--	15.41	--	--
Context-pipelining-3	--	--	22.37	33.03
Context-pipelining-4	--	--	--	29.88

Context-pipelining-3 & 4 are the mixing scheme of Context-pipelining-1 & 2, in which each stage is replicated using multi-processing. For example, context-pipelining-3 uses 4 MEs, in which 2 MEs are allocated for the first stage of context-pipelining-1 and the remaining 2 MEs are allocated for the second stage of context-pipelining-1, It is clear that both multi-processing and context-pipelining-3 & 4 can support the OC-192 line rate with four MEs and six MEs respectively on the Intel IXP2800. However multi-processing is preferable for Bitmap-RFC algorithm because of the dynamic nature of the workload.

6.7 Latency Hiding

Table 9 reports the performance impact of various latency hiding techniques. The MicroengineC compiler provides only one switch to turn latency hiding optimizations on or off. We reported the combined effects after applying those latency hiding techniques. The MicroengineC compiler can schedule ALU instructions into the delay slots of a conditional/unconditional branch instruction and a SRAM/DRAM memory instruction.

Table 9. Improvement from latency hiding techniques (Mpps)

	1 ME	2 MEs	4 MEs	8 MEs
Overlapped	6.54	12.85	25.65	33.35
Without overlapped	6.10	12.13	23.94	33.10
Improvement	7.11%	7.15%	7.14%	0.64%

By performing static profiling, we found that seventeen ALU instructions were scheduled into delay slots. On average, we obtained a performance improvement of approximately 7.13% by applying the latency hiding techniques.

7. PROGRAMMING GUIDANCE ON NPU

We have presented Bitmap-RFC implementations and analyzed performance impacts on the Intel IXP2800. Based on our experiences, we provide the following guidelines for creating an efficient network application on an NPU.

1) Compress data structures and store them in SRAM whenever possible to reduce memory access latency.

2) Multi-processing is preferred to parallelize network applications rather than context pipelining because the former is insensitive to workload balance. Unless the workload can be statically determined, use a combination of both to help distribute loads among different processing stages fairly.

3) In general, the NPU has many different shared resources, such as command and data buses. Pay attention to how those shared resource are used because they might become a bottleneck in algorithm implementation.

4) The NPU supports powerful bit-manipulation instructions. Select appropriate instructions to meet the application needs without waiting for the compiler automation support.

5) Use compiler optimizations to schedule ALU instructions to fill the delay slots to hide latency whenever possible.

8. CONCLUSIONS AND FUTURE WORK

This paper proposed a high-speed packet classification algorithm Bitmap-RFC and its efficient implementation on the Intel IXP2800. We studied the interaction between the parallel algorithm design and architecture mapping to facilitate efficient algorithm implementation on the NPU architecture. We experimented with an architecture-aware design principle to guarantee the high-performance of the resulting algorithm. Furthermore, we investigated the main software design issues that have most dramatically performance impacts on networking applications. Based on detailed simulation and performance analysis, we identified the limits of classification algorithm on an IXP2800. We effectively exploited the thread-level parallelism on many-core and multithreaded architectures to enable an efficient algorithm mapping.

Our experiences show that developing networking applications on a many-core and multithreaded architecture requires applying a system method to address the performance bottleneck. The architecture-aware method promoted in this paper advocates of considering the architectural features and constraints in the algorithm development phases as early as in algorithm design. Furthermore, in each development phase from algorithm design to algorithm implementation, the design decisions made should be based on the application characteristics as well as the architectural features. For example, the bitmap-RFC relies on the following application characteristics and the architecture features to achieve 10Gbps speed:

- the RFC algorithm can be easily parallelizable;
- uncompression of the bitmap compressed tables can be efficiently performed on the IXP2800 using special bit-manipulation instructions;
- execution of the multiple outstanding memory accesses can be overlapped with other useful computation in the multi-threaded architecture.

By exploiting the application characteristics and the architectural features, the high-performance classification algorithm bitmap-RFC has been developed on the IXP2800.

Our performance analysis indicates that we need spend more effort on eliminating various hardware performance bottlenecks, such as the SARM and DRAM buses. In addition, how to select an appropriate size of sub-CPT to make Bitmap-RFC run faster requires more study. We will do more research along these two directions.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This work was supported by the Intel China IXA University Program, the National Natural Science Foundation of China, and the Anhui Province-MOST Co-Key Laboratory of High Performance Computing and Its Application.

REFERENCES

[1] Agere, “Network Processors”, http://www.agere.com/telecom/network_processors.html.

[2] J. R. Allen, B. M. Bass, C. Basso, R. H. Boivie, J. L. Calvignac, G. T. Davis, L. Frelechoux, M. Heddes, A., et al., “IBM PowerNP Network Processor: Hardware, Software, and Applications”, IBM J. Res. & Dev., Vol. 47 No. 2/3 MARCH/MAY 2003.

[3] AMCC, “Network Processors”, https://www.amcc.com/MyAMCC/jsp/public/browse/controller.jsp?networkLevel=COMM&superFamily=NETP.

[4] Avici, “Avici Intros Multiservice Line Cards”, http://www.lightreading.com/document.asp?doc_id=34665&site=supercomm

[5] F. Baboescu, S. Singh, and G. Varghese, “Packet Classification for Core Routers: Is there an alternative to CAMs”, Technical Report, University of California, San Diego, 2003.

[6] F. Baboescu and G. Varghese, “Scalable Packet Classification”, in Proc. of ACM SIGCOMM, 2001, pp.199-210.

[7] Cisco Systems, “Cisco CRS-1 Carrier Routing System”, http://www.cisco.com/en/US/products/ps5763/

[8] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, “Small Forwarding Tables for Fast Routing Lookups”, in Proc. of ACM SIGCOMM ’97, Cannes, France, 1997, pp.3-14.

[9] W. Eatherton, G Varghese, and Z Dittia, “Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates”, in Proc. of ACM SIGCOMM on Computer Communication Review, Vol. 34, Issue 2, Apr. 2004, pp.97-122.

[10] Freescale, “C-Port Network Processors”, http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=02VS0lDFTQ3126.

[11] P. Gupta and N. McKeown, “Packet Classification Using Hierarchical Intelligent Cuttings”, IEEE Micro, Vol. 20, No. 1, Jan.-Feb. 2000, pp.34-41.

[12] P. Gupta and N. McKeown, “Packet Classification on Multiple Fields”, in Proc. of ACM SIGCOMM, Computation Communication Rev., Vol. 29, Sep. 1999, pp.147-160.

[13] Xianghui Hu, Xinan Tang, and Bei Hua, “A High-performance IPv6 Forwarding Algorithm for a Multi-core and Multithreaded Network Processor”, in Proc. of ACM PPoPP’06, Mar. 2006, pp.168-177.

[14] Huawei, “Huawei Launches NetEngine80 Core Router At Networld Interop 2001 Exhibition in US”, http://www.huawei.com/news/view.do?id=88&cid=-1001

[15] Intel, “IXP2XXX Network Processors”, http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm.

[16] M. Kounavis et al., “Directions in Packet Classification for Network Processors”, in Proc. of Second Workshop on Network Processors (NP2), Feb. 2003.

[17] C. Kulkarni, M. Gries, C. Sauer, and K. Keutzer, “Programming Challenges in Network Processor Deployment”, in Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded System, San Jose, USA, 2003, pp.178-187.

[18] T. V. Lakshman and D. Stiliadis, “High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching”, in Proc. of ACM SIGCOMM98, Sep. 1998, pp. 191-202.

[19] T. Sherwood, G. Varghese and B. Calder, “A Pipelined Memory Architecture for High Throughput Network Processors”, in Proc. of ACM ISCA’03, 2003.

[20] S. Singh, F. Baboescu, G. Varghese, and Jia Wang, “Packet Classification Using Multidimensional Cutting”, in Proc. of ACM SIGCOMM’03, ACM Press, 2003, pp.213-224.

[21] E. Spitznagel. “Compressed Data Structures for Recursive Flow Classification”, Technical Report, WUCSE-2003-65, May 2003.

[22] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel, “Fast and Scalable Layer Four Switching”, in Proc. of ACM SIGCOMM’98, Sep. 1998, pp. 203-14.

[23] Xinan Tang and Guang R. Gao, “Automatically Partitioning Threads for Multithreaded Architectures”, in Journal of Parallel Distributed Computing, 1999,58(2) pp.159-189.

[24] Xinan Tang and Guang R. Gao, “How hard is thread partitioning and how bad is a list scheduling based partitioning algorithm?", in Proc. of the tenth annual ACM symposium on Parallel Algorithms and Architectures, pp. 159-189, 1998.

[25] Xinan Tang, J. Wang, K. Theobald, and Guang R. Gao, “Thread partitioning and scheduling based on cost model”, in Proc. of the ninth annual ACM symposium on Parallel Algorithms and Architectures, pp. 272-281, 1997.

[26] D. E. Taylor and J. S. Turner, “ClassBench: A Packet Classification Benchmark”, Technical Report, WUCSE-2004-28, Department of Computer Science & Engineering, Washington University in Saint Louis, May 2004.

[27] Yaxuan Qi and Jun Li, “Towards Effective Packet Classification”, in Proc. of IASTED Conference on Communication, Network, and Information Security (CNIS), 2006.