High-Performance Packet Classification Algorithm for Multithreaded IXP Network Processor

DUO LIU
Southwest University of Science and Technology, University of Science and Technology of China, and Suzhou Institute for Advanced Study of USTC
ZHENG CHEN
University of Science and Technology of China
BEI HUA
University of Science and Technology of China and Suzhou Institute for Advanced Study of USTC
NENGHAI YU
University of Science and Technology of China
and
XINAN TANG
Intel Corporation

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 10 Gbps 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 exploit the multithreaded architectural features in various algorithm development stages from algorithm design to algorithm implementation. As a result, Bitmap-RFC strikes a good balance between speed and space. It can significantly

Authors' addresses: Duo Liu, School of Computer Science and Technology, Southwest University of Science and Technology, Mianyang, P.R. China, 621010; email: liuduo@swust.edu.cn; Zheng Chen, Nenghai Yu, and Bei Hua, Department of Computer Science and Technology, University of Science and Technology of China, Hefei, P.R. China, 230027; email: {jzchen, ynh}@ustc.edu.cn; email: bhua@ustc.edu.cn; Xinan Tang, Intel Compiler Lab, Intel Corporation, Santa Clara, California, 95054; email: xinan.tang@intel.com. The first author did this work when he was a graduate student at USTC and stayed at Suzhou institute of USTC.

keep both high classification speed and reduce memory space consumption. 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 10 Gbps 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

General Terms: Performance, Algorithms, Design, Experimentation

Additional Key Words and Phrases: Network processor, packet classification, architecture, multithreading, thread-level parallelism, embedded system design

ACM Reference Format:

Liu, D., Chen, Z., Hua, B., Yu, N., and Tang, X. 2008. High-Performance packet classification algorithm for multithreaded IXP network processor. ACM Trans. Embedd. Comput. Syst. 7, 2, Article 16 (February 2008), 25 pages. DOI = 10.1145/1331331.1331340 http://doi.acm.org/10.1145/1331331.1331340

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 10 Gbps 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 [Intel], Freescale [Freescale], AMCC [Amcc], and Agere [Agere] 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 [Avici; Cisco Systems; Huawei]. In addition, an NPU is a special designed many-core and multi-threaded system targeted for network applications. An NPU-based classification algorithm is worth 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 [Srinivasan et al. 1998], bit vector linear search [Baboescu and Varghese 2001; Lakshman and Stiliadis 1998], cross-producting [Srinivasan et al. 1998]

and recursive flow classification (RFC) [Gupta and McKeown 1999], and decision-tree approaches [Gupta and McKeown 2000; Singh et al. 2003]. All these algorithms focus on striking a 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 parallel architectures, especially on a many-core and multithreaded NPU architecture. New efforts are, therefore, required to design parallel packet classification algorithms for a 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 development stage, from algorithm design to 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 [Gupta and McKeown 1999] is, thus 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 is to traverse the tree from the root to the leaves. The traversal of the tree is easy to parallelize, because (1) nodes on the same level can potentially run in parallel; and (2) nodes on different levels can also run in parallel.

On a multithreaded architecture, latency hiding can be utilized in three ways [Tang et al. 1997; Tang and Gao 1998, 1999]. 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.

In contrast to the performance measurement methodology used by other classification algorithms in which performance is predicted by counting the number of memory accesses, we measure classification performance in throughput. Even though the number of memory accesses of an algorithm is one of the

most important metrics to measure performance, it would be inaccurate to use it solely to measure performance, especially on a multithreaded architecture [Tang and Gao 1999]. For example, two parallel memory accesses can be issued consecutively on a multithreaded architecture and thus the latency of the first memory access can be partially hidden by that of the second memory access [Tang and Gao 1998]. The latencies incurred by those two parallel memory accesses are thus much smaller than the sum of latencies caused by two sequential memory accesses. In addition, the latency of one memory access can be overlapped with another thread's execution. Therefore, the latency hiding ability of multithreading makes performance measurement harder on a multithreaded architecture [Tang et al. 1997]. We believe throughput should be used in the future as a means to compare algorithmic classification performance.

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 multiprocessing research are not directly applicable to such system-on-chip (SOC)-based many-core and multithreaded architectures because of their specific processing requirements [Allen et al. 2003; Kulkarni et al. 2003]. 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 [Degermark et al. 1997] 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 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 [Agere; Amcc; Freescale]. 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 10 Gbps.
- 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 10 Gbps 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 [Baboescu et al. 2003; Baboescu and Varghese 2001; Gupta and McKeown 1999, 2000; Lakshman and Stiliadis 1998; Singh et al. 2003; Srinivasan et al. 1998; Qi and Li 2006]. Below, we mainly compare *algorithmic* classification schemes, especially from the NPU implementation point of view.

Trie-based algorithms, such as grid-of-tries [Srinivasan et al. 1998], 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 [Baboescu and Varghese 2001; Lakshman and Stiliadis 1998] 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) [Gupta and McKeown 2000] 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 [Singh et al. 2003] improves upon HiCuts, in which each node represents a decision point in the multidimensional hypercube. HyperCuts attempts to minimize the depth of the decision tree by extending the single-dimensional into a multidimensional search. 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 [Gupta and McKeown 1999], which is a generalization of cross-producting [Srinivasan et al. 1998], 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 [Spitznagel 2003], 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 [Degermark et al. 1997; Eatherton et al. 2004] and IPv6 forwarding [Hu et al. 2006]. Recently, it was applied to classification [Spitznagel 2003], which is the closest in spirit to ours in that all use bitmaps to compress redundancies appeared in the internal tables. However, the 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 [Degermark et al. 1997] algorithm introduces a summary array to precompute the number of bits set in the bitmap and, thus, it needs an extra memory access per trie-node to search the compressed table. Our 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. We first applied this technique in IPv6 forwarding [Hu et al. 2006] and showed it is very effective to use such a compression technique on Intel IXP architecture. This paper extends our previous work and applies the same bitmap-compression technique on the classification tables. One of the interesting results is that the bitmap technique has the same compression effect on both forwarding and classification tables.

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 ith field of rule R_j, referred to as $R_j[i]$, is a regular expression pertaining to the ith 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

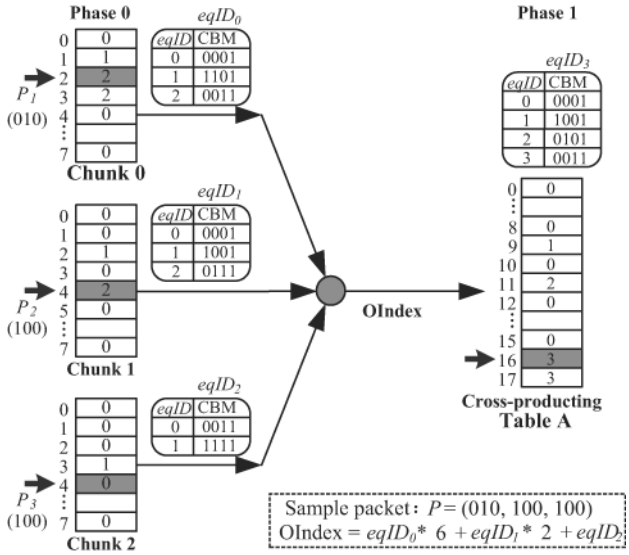

Fig. 1. A two-phase RFC reduction tree.

Table I. 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

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.

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 large. The main idea of RFC is to split such a one-time mapping into multiphase mapping in order to reduce a larger search space into multiple smaller ones. Each mapping phase is called a *reduction* and the data structure formed by multiphase mapping is called a *reduction tree*. After multiphase mapping, S-bit CF-string is mapped to a T-bit ($T << S$) space.

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 I, 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, which is concatenated by 3 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 and compute the index of cross-producting table $\boldsymbol{A}$ by $\mathbf{Chunk_0[2]*3*2} + \mathbf{Chunk_1[4]*2} + \mathbf{Chunk_2[4]}$, which is 16;
2. Use CPT[16], which is 3, as an index to search $eqID_3$. 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 10 Gbps or higher on Intel IXP 2800.

Similar to the scheme used in [Gupta and McKeown 1999], we use a three-phase reduction tree for classification.

- Phase 0 contains 6 chunks: chunk 0 uses the high 16 bits of source IP address, chunk 1 uses the low 16 bits of source IP address, chunk 2 uses the high 16 bits of destination IP address, chunk 3 uses the low 16 bits of destination IP address, chunk 4 uses source port, and chunk 5 uses destination port;
- Chunk 0 (CPT X) of phase 1 is formed by combining chunk 0, 1, 4 of phase 0;
- Chunk 1 (CPT Y) of phase 1 formed by combining chunk 2, 3, 5 of phase 0;
- Chunk 0 (CPT Z) of phase 2 is formed by combining the two chunks of phase 1.

The two index of phase 1 is formed by

$$i_{10} = e_{00} \times m_{01} \times m_{04} + e_{01} \times m_{04} + e_{04} \quad (1)$$

$$i_{11} = e_{02} \times m_{03} \times m_{05} + e_{03} \times m_{05} + e_{05} \quad (2)$$

The index of phase 2 is formed by

$$i_{20} = e_{10} \times m_{11} + e_{11} \quad (3)$$

where e_{ij} donates the *eqID* retrieved from previous phase and m_{ij} donates the unique *eqID* numbers of phase i chunk j.

4. BITMAP-RFC ALGORITHM

4.1 Algorithm Description

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

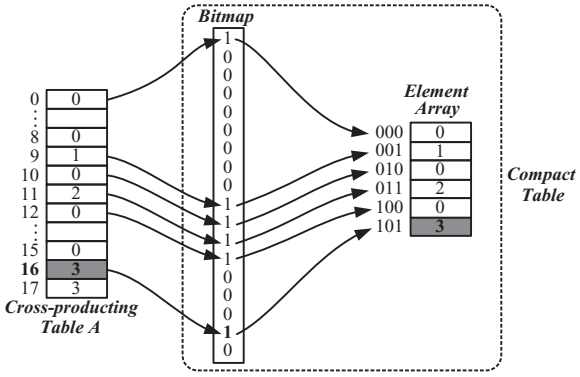

Fig. 2. Illustration of Bitmap-RFC algorithm.

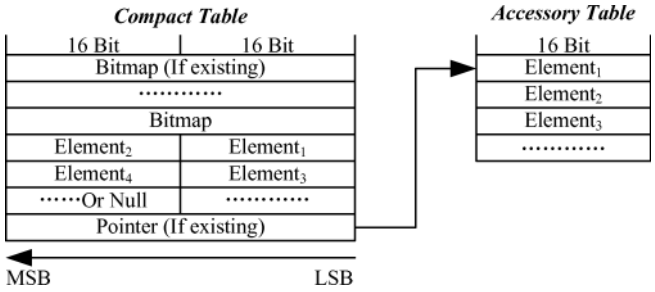

Fig. 3. Data structure for Bitmap-RFC algorithm.

denote this data structure in the rest of this paper. Only one entry of the compact table is illustrated in this figure because of space limitations. 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.

The reason for putting bitmap and element array together is to obtain efficient memory access by utilizing a block-move 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.

```
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 (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
```

Fig. 4. Pseudocode for Bitmap-RFC search operation.

4.3 Bitmap-RFC Lookup

Figure 4 gives the pseudocode 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 eight elements of 16-bits each. The entry size of compact table is five words, which equals to the sum of Bitmap and element's size. That means one memory access should fetch five 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 eight 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 value of Bitmap[0] is 1, 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 function **GetBitPos** (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 13–19). If an IE being searched is in the element array (no greater than eight entries as shown in line 13), the result is returned from line 14; otherwise,

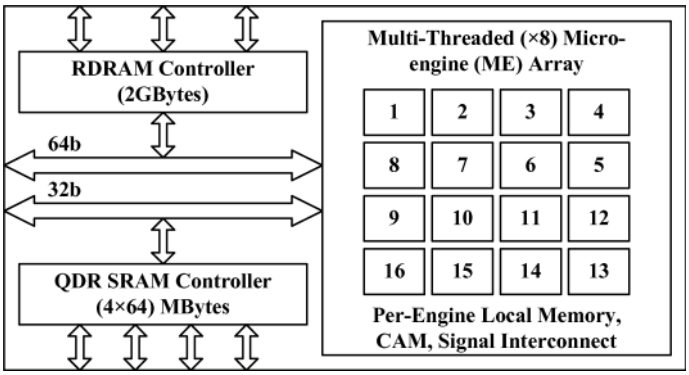

Fig. 5. IXP2800 component diagram without I/O interfaces.

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 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.

5. NPU-AWARE DESIGN AND IMPLEMENTATION

Figure 5 draws the components of the Intel IXP2800 [Intel], in which 16 Micro-engines (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. There are two clusters of MEs, ME_{1-8} and ME_{9-16}. Each cluster can use a separate bus to access 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.

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 four words containing, at most, eight elements, because the number of bits set in a sub-CPT is most likely no greater than eight. This is an observation we have noticed for the classification rules we have experimented on (see Figure 8). 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 three 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.

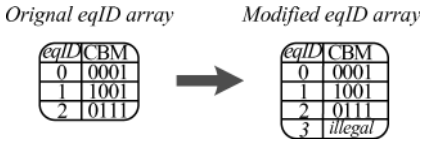

Fig. 6. Illustration of extended eqID array.

5.3 Multiplication Elimination

Although NPUs contain special instructions to facilitate packet processing, they may lack efficient implementation of the multiply instruction. For example, a 32 bit multiply instruction will cost five cycles in Intel IXP2800. Furthermore, a multiply instruction cannot be scheduled into a delay slot. At 10 Gbps and ME at 1.4 GHz on IXP2800, each packet must be processed within 60 cycles. The 25-cycles of five multiply instructions takes 41.6% of packet processing budget. Therefore, it is benefit to perform compiler strength reduction, which is to replace a multiply instruction with a shift one.

In order to substitute the multiply with shift in the algorithm, we must extend the size of some *eqID* array to $2^{\lceil \log m \rceil}$, where m is the original size of this *eqID* array, i.e., the size of *eqID* array will be aligned to the power of 2. The new *eqID* array has extended *eqID*s and each with a special CBM, *illegal* CBM, just as illustrated in Figure 6. Anything that intersects with the *illegal* entry will produce an *illegal* entry. Even though some *illegal* entries are added into CPT tables, if those *illegal* entries are consecutive located, the Bitmap compression can effective eliminate such redundancy, and it will not cause any significant increase of memory space. Note, also, that this modification will only affect the second and third level of CPT tables. The first level CPT tables are still the same. After this modification, the table index can be calculated by the new Eqs. (4)–(6) with shift operations.

$$i_{10} = e_{00} \times 2^{\lceil \log m_{01} \rceil} \times 2^{\lceil \log m_{04} \rceil} + e_{01} \times 2^{\lceil \log m_{04} \rceil} + e_{04} \quad (4)$$

$$i_{11} = e_{02} \times 2^{\lceil \log m_{03} \rceil} \times 2^{\lceil \log m_{05} \rceil} + e_{03} \times 2^{\lceil \log m_{05} \rceil} + e_{05} \quad (5)$$

$$i_{20} = e_{10} \times 2^{\lceil \log m_{11} \rceil} + e_{11} \quad (6)$$

5.4 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.

Table II. 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'

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.

We experimented on various data allocation schemes for the fourth setting, however, only the one shown in Table II 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 four 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 as 20 bytes (less than 64 bytes), memory vectorization optimization can be applied to significantly reduce the number of SRAM accesses.

5.5 Task Partitioning

There are two general ways to partition tasks onto multiple MEs on the Intel IXP2800: *multiprocessing* and *contextpipelining*. Multiprocessing involves two parallelizing techniques. First, multithreading is applied to a task allocated to one ME. In an Intel IXP2800, a maximum of eight threads can be used per ME. Second, 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 is easily implemented. In addition, the workloads are easier to balance. 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 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 multiprocessing scheme may achieve higher performance than context pipelining. The simulation results shown in Section 6.7 confirm this prediction.

5.6 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 lines 3 and 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 [Tang and Gao 1999].

6. SIMULATION AND PERFORMANCE ANALYSIS

Because of privacy and commercial secrets, it is hard for us to access sufficient real classifiers. Luckily, the characteristics of real classifiers have been analyzed in [Baboescu et al. 2003; Kounavis et al. 2003; Taylor and Turner 2005]. Therefore, we could construct the synthetic ones for the core router accordingly [Singh et al. 2003]. In order to measure the performance impact of the Intel IXP2800 on the Bitmap-RFC algorithm, we experiment with the following implementations discussed in Sections 6.2–6.9.

6.1 Experimental Setup

As mentioned in Section 3, 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 four-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 (nine times). As a result, more rules only affect the memory space needed. Therefore, three classifiers were constructed for the experiments presented in Sections 6.3–6.8. They are CL#1, CL#2, and CL#3, and each has 1000, 2000, and 3000 rules, respectively.

Table III. Comparison of Memory Requirements

	Memory requirement of CPT and CCPT (MB)						Total memory requirement (MB)	
Num. of Rules	X	X'	Y	Y'	Z	Z'	RFC	Bitmap-RFC
5700	59.2	18.5	25.8	8.1	***64.1***	16.0	149.9	43.4
8050	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***

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 five long words.

We use the minimal packets as the worst-case input [Sherwood et al. 2003]. 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 25.5 Mpps (million packets per second) is required to achieve the OC-192 line rate. All experiments were done on a cycle-accurate simulator, except the experiments described in Section 6.9, which were carried out on a hardware platform.

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

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

When the number of rules exceeds 5700, RFC cannot be implemented on IXP2800, since the memory requirement of table Z is bigger than 64 MB. In addition, the total memory requirement of RFC will be larger than 256 MB when 8050 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 64 MB. In addition, the total memory requirement of Bitmap-RFC will be bigger than 256 MB when 17,000 rules are used.

Since SRAM space required for Z' bank is over 64 MB when 12,000 rules are used, the corresponding CPT must be split into two subtables appropriately to fit into two SRAM banks. Such a split can be done 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 Classifying Rates of RFC and Bitmap-RFC

Table IV shows the performance comparison between RFC and Bitmap-RFC. We use million packets per second to measure performance. Both algorithms require three MEs to reach OC-192 speed. The classifying rates of Bitmap-RFC is very close to that of RFC, and the difference between two classification

Table IV. Classifying Rates (Mpps) of RFC versus Bitmap-RFC

Classifier	RFC		Bitmap-RFC	
	Num. of MEs	Rates	Num. of MEs	Rates
CL#1	3	27.17	3	26.65
CL#2	3	26.11	3	25.74
CL#3	3	26.58	3	25.77

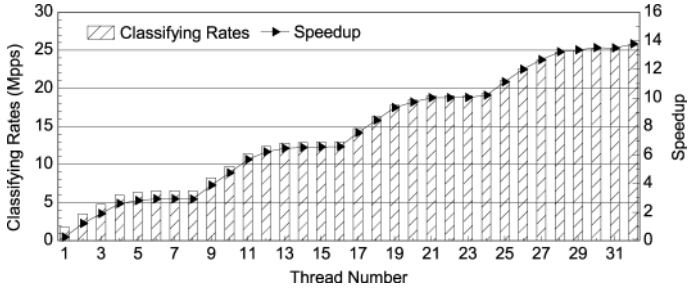

Fig. 7. Bitmap-RFC classifying rates and relative speedups.

rates is less than 3%. This is partially because we use the POP_COUNT instruction to uncompress the classification table. Compared to the original RFC the three-cycle latency of POP_COUNT is indeed an overhead for each table access. However, this latency can be hidden by the memory accesses of other threads. Therefore, its performance impact can be minimized by executing multiple threads simultaneously.

Based on this result, we can conclude that Bitmap-RFC strikes a good balance between speed and space. It can not only keep high classification speed but also reduce memory space significantly.

6.4 Relative Speedups

Figure 7 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 25.65 Mpps for 32 threads. The reason of the sublinear speedup is partially caused by the saturation of command request FIFOs and SRAM buses. The SRAM bus behavior is illustrated in Table V, in which the FIFO fullness ratio can be used to indicate the degree of saturation.

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 eight-thread mode, a read memory request stays almost four times longer in the FIFO than it does in the one-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

Table V. 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

Table VI. 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

Table VII. Classifying Rates (Mpps) of POP_COUNT versus 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%

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 VI 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.

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.

6.5 Instruction Selection

Table VII 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 8 MEs, because the utilization rate of four SRAM controllers has reached 96%, in both cases and the SRAM bandwidth becomes a new performance bottleneck, which

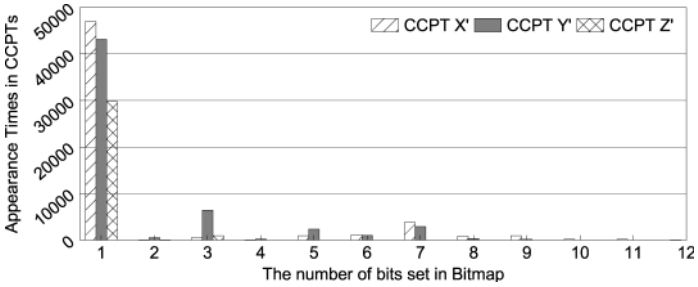

Fig. 8. Distribution of the number of bits set in Bitmap.

Table VIII. 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

shadows other performance improvement factors. The same impact can also be observed in other performance tables reported in Sections 6.7 and 6.8.

The lower classifying rate of **FFS** is because computational time of **Position-Num** 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 with, 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 8, 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 versus FFS should be even higher.

6.6 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 VIII shows the simulation results. We found out:

- 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

Table IX. Classifying Rates (Mpps) of Multiprocessing versus Context Pipelining

	2 MEs	3MEs	4 MEs	6 MEs
Multiprocessing	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

rate of a single SRAM controller is up to 98% at 4 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 up to 8 MEs. In addition, 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.

6.7 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 IX shows the simulation results using different task allocation policies.

Context pipelining-3 and -4 are the mixing scheme of context pipelining-1 and -2, in which each stage is replicated using multiprocessing. 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 multiprocessing and context pipelining-3 and -4 can support the OC-192 line rate with 4 and 6 MEs, respectively, on the Intel IXP2800. However, multiprocessing is preferable for Bitmap-RFC algorithm because of the dynamic nature of the workload.

Table X. Improvement from Latency Hiding Techniques (Mpps)

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

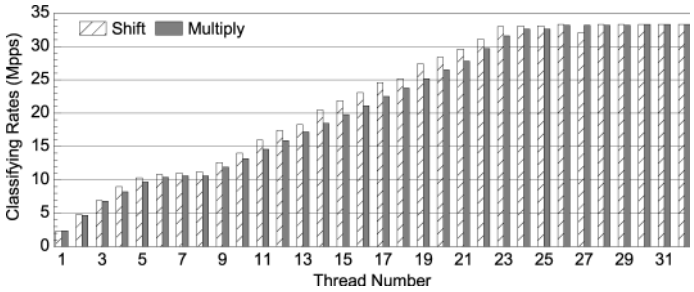

Fig. 9. Classifying rates of shift versus multiply.

6.8 Latency Hiding

Table X 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.

By performing static profiling, we found that 17 ALU instructions were scheduled into delay slots. On average, we obtained a performance improvement of approximately 7.13% by applying the latency-hiding techniques.

6.9 Multiplication Elimination

Figure 9 shows the classifying rates of the worst-case input packets by using two different scenarios: (1) compute the index by shift instructions; (2) by multiply instructions, respectively. These results were gathered from hardware experiments.

The classifying rate of shift is higher than that of multiply when the threads number is up to 24. The performance improvement of shift over multiply can be high, as much as 10%. The best cases are incurred when 14 and 15 threads were used. In these two situations, we can obtain as much as 10% performance gain. The exception is when the threads number is greater than 24, because the utilization of the SRAM controllers has reached 96% in both cases. Thus, the SRAM references become a new performance bottleneck, which shadows other performance factors. The same situation was observed in previous subsections.

Please note that the hardware classification speeds are higher than those of the simulation based. A partial reason is that the input traces are different and there is some discrepancy between the cycle-accurate simulator and real

silicon in terms of entire chip-level simulation, especially in memory system simulation.

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. Multiprocessing 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 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 we promoted advocates of considering the architectural features and constraints in the various algorithm development stages. Furthermore, in each development stage from algorithm design to algorithm implementation, the design decision exploration 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 10 Gbps:

- 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 multithreaded 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 (1) the IXP memory system becomes a hardware performance bottleneck after we enhance the ME performance; (2) the appropriate size of sub-CPT affects the Bitmap-RFC performance. We will do more research along these two directions.

In addition, as NPU becomes widely used in practice, parallelizing network applications on NPU becomes even more important, since programming the multithreaded architecture is a new venture for most of networking programmers. We will continue to find effective ways to map networking applications onto the many-core and multithreaded architecture.

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

AGERE. Network Processors. http://www.agere.com/telecom/network_processors.html.

ALLEN, J. R., BASS, B., ET AL. 2003. IBM PowerNP Network Processor: Hardware, Software, and Applications. *IBM J. Res. & Dev.*, *47*, 2/3 (Mar./May).

AMCC. Network Processors. https://www.amcc.com/MyAMCC/jsp/public/browse/controller.jsp?networkLevel=COMM&superFamily=NETP.

AVICI. Avici Intros Multiservice Line Cards. http://www.lightreading.com/document.asp?doc_id=34665&site=supercomm.

BABOESCU, F. AND VARGHESE, G. 2001. Scalable packet classification. In *Proceedings of ACM SIGCOMM'01*. San Diego, California. 199–210.

BABOESCU, F., SINGH, S., AND VARGHESE, G. 2003. Packet classification for core routers: Is there an alternative to CAMs? In *INFOCOM 2003, Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 1, 53–63.

CISCO SYSTEMS. Cisco CRS-1 Carrier Routing System. http://www.cisco.com/en/US/products/ps5763/.

DEGERMARK, M., BRODNIK, A., CARLSSON, S., AND PINK, S. 1997. Small forwarding tables for fast routing lookups. In *Proceedings of ACM SIGCOMM'97*, Cannes, France. 3–14.

EATHERTON, W., VARGHESE, G., AND DITTIA, Z. 2004. Tree bitmap: Hardware/software IP lookups with incremental updates. In *ACM SIGCOMM Computer Communication Review 34*, 2, (Apr.), 97–122.

FREESCALE. C-Port Network Processors. http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=02VS0lDFTQ3126.

GUPTA, P. AND MCKEOWN, N. 1999. Packet classification on multiple fields. In *Proceedings of ACM SIGCOMM'99*. Cambridge, MA. 147–160.

GUPTA, P. AND MCKEOWN, N. 2000. Classifying packets with hierarchical intelligent cuttings. In *IEEE Micro 20*, 1, (Jan./Feb.), 34–41.

HU, X. H., TANG, X. N., AND HUA, B. 2006. High-performance IPv6 forwarding algorithm for a multi-core and multithreaded network processor. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. Mar. 168–177.

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

INTEL. IXP2XXX Network Processors. http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm.

KOUNAVIS, M., ET AL. 2003. Directions in packet classification for network processors. In *Proceedings of Second Workshop on Network Processors (NP2)*.

KULKARNI, C., GRIES, M., SAUER, C., AND KEUTZER, K. 2003. Programming challenges in network processor deployment. In *Proceedings of the 2003 International Conference on Compilers, Architecture, and Synthesis for Embedded System*. San Jose, CA. 178–187.

LAKSHMAN, T. V. AND STILIADIS, D. 1998. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of ACM SIGCOMM'98*. Vancouver, British Columbia, Canada. 203–214.

QI, Y. X. AND LI, J. 2006. Towards effective packet classification. In *Proceedings of IASTED Conference on Communication, Network, and Information Security (CNIS)*.

SHERWOOD, T., VARGHESE, G., AND CALDER, B. 2003. A pipelined memory architecture for high throughput network processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ACM ISCA'03)*. San Diego, CA. 288–299.

SINGH, S., BABOESCU, F., VARGHESE, G., AND WANG, J. 2003. Packet classification using multidimensional cutting. In *Proceedings of ACM SIGCOMM'03*. Karlsruhe, Germany. 213–224.

SPITZNAGEL, E. 2003. Compressed Data Structures for Recursive Flow Classification. http://www.cse.seas.wustl.edu/Research/FileDownload.asp?295.

SRINIVASAN, V., SURI, S., VARGHESE, G., AND WALDVOGEL, M. 1998. Fast and scalable layer four switching. In *Proceedings of ACM SIGCOMM'98*. Vancouver, British Columbia, Canada. 191–202.

TANG, X. N. AND GAO, G. R. 1998. How "hard" is thread partitioning and how "bad" is a list scheduling based partitioning algorithm? In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures (SPAA'98)*. Puerto Vallarta, Mexico. 130–139.

TANG, X. N. AND GAO, G. R. 1999. Automatically partitioning threads for multithreaded architectures. In *Journal of Parallel Distributed Computing 58*, 2 (Aug.), 159–189.

TANG, X. N., WANG, J., THEOBALD, K., AND GAO, G. R. 1997. Thread partitioning and scheduling based on cost model. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*. Newport, RI. 272–281.

TAYLOR, D. E. AND TURNER, J. S. 2005. ClassBench: A packet classification benchmark. In *Proceedings of IEEE INFOCOMM'05*. Miami, FL. 2068–2079.

Received January 2007; revised June 2007; accepted June 2007