Scalable Packet Classification Using Interpreting A Cross-platform Multi-core Solution

[1]Haipeng Cheng, [2]Zheng Chen, [3]Bei Hua
[1,3]Department of Computer Science and Technology
[2]Department of Electronic Engineering and Information Science
University of Science and Technology of China
Hefei, Anhui, 230027, China
{[1]hpcheng, [2]jzchen}@mail.ustc.edu.cn
[3]bhua@ustc.edu.cn

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

Abstract

Packet classification is an enabling technology to support advanced Internet services. It is still a challenge for a software solution to achieve 10Gbps (line-rate) classification speed. This paper presents a classification algorithm that can be efficiently implemented on a multi-core architecture with or without cache. The algorithm embraces the holistic notion of exploiting application characteristics, considering the capabilities of the CPU and the memory hierarchy, and performing appropriate data partitioning. The classification algorithm adopts two stages: searching on a reduction tree and searching on a list of ranges. This decision is made based on a classification heuristic: the size of the range list is limited after the first stage search. Optimizations are then designed to speed up the two-stage execution. To exploit the speed gap (1) between the CPU and external memory; (2) between internal memory (cache) and external memory, an interpreter is used to trade the CPU idle cycles with demanding memory access requirements. By applying the CISC style of instruction encoding to compress the range expressions, it not only significantly reduces the total memory requirement but also makes effective use of the internal memory (cache) bandwidth. We show that compressing data structures is an effective optimization across the multi-core architectures.

We implement this algorithm on both Intel IXP2800 network processor and Core 2 Duo X86 architecture, and experiment with the classification benchmark, ClassBench. By incorporating architecture-awareness in algorithm design and taking into account the memory hierarchy, data partitioning, and latency hiding in algorithm implementation, the resulting algorithm shows a good scalability on Intel IXP2800. By effectively using the cache system, the algorithm also runs faster than the previous fastest RFC on the Core 2 Duo architecture.

Categories and Subject Descriptors C.1.4 [**Processor Architectures**]: Parallel Architectures; C.2.6 [**Computer-communication Networks**]: Internetworking – *Routers*; D.2.2 [**Software Engineering**]: Design Tools and Techniques;

General Terms Algorithms, Experimentation, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPoPP'08 February 20-23, 2008, Salt Lake City, Utah, USA.
Copyright © 2008 ACM 978-1-59593-960-9/08/0002…$5.00.

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

1. Introduction

Packet classification is an enabling technology to support advanced Internet services such as network security, QoS provisioning, traffic policing, and virtual private network. The following IP header fields: the source and destination addresses, source and destination ports, and protocol type, are generally used to classify packets into flows for appropriate processing. As more demand for triple-play (voice, video, and data) services arises, the pressure to perform fast packet classification becomes 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.

There are two types of the shared memory based multi-core architectures: (1) the general-purpose multi-core with two or more levels of cache memories such as Intel Core 2 Duo [2], Sun Niagara [3], Cavium Octeon, and RMI XLR; (2) the special-purpose multi-core without cache such as Intel IXP for networking [9] and IBM Cell for gaming [8]. However, each IXP's microengine and Cell's synergistic processor element has much faster on-chip memory, which the programmer can directly manage. The networking industry has been using both types of multi-cores in blade-server firewalls and in routers/switches. As the trend toward multi-core deployment in the networking space becomes strong, software design issues for network applications are worthy of further study. Particularly an across-platform classification algorithm is actively sought in order to realize multi-core potential.

In general, a good packet classification algorithm focuses on striking a balance between space and speed to achieve optimal algorithmic performance. However, little work has been done in parallelizing these algorithms on the multi-core architectures. Furthermore, most of the existing classification algorithms were not designed for the multi-cores, and cannot be directly applied. New efforts are therefore required to design parallel packet classification algorithms for the multi-cores, which normally provides either hardware-assisted multithreading [3][9] to hide memory latency [20][21][22] or cache memories to reduce memory latency.

In this paper, we propose an architecture-aware classification algorithm that exploits both the application and the multi-core architectural features to reduce memory-access times as well as

hide the memory-access latency. In particular, we adopt a system approach in designing an efficient classification algorithm and finding the best solution in each algorithm decision point, from algorithm design to algorithm implementation.

Even though there are many multi-core architectures, two challenges are common in any multi-core software design. First, as the speed gap between CPU and memory grows, it becomes increasingly difficult to keep a single CPU busy, and it is even more difficult to maintain multiple CPU cores running at full speed. Second, as the memory hierarchy becomes more complex, it gets harder to optimize data layout for the various levels of cache memories in a single CPU, and it is even harder to optimize data layout for the distributed cache memories in multi-cores. Cache compression is one way to improve the effectiveness of cache memories [1]. We explore the compression idea at the application level to address the two software design issues for the multi-cores.

Firstly, compression enables cache (on-chip memories) to store more data, and therefore significantly reduce the memory access latency. For example, moving data from DRAM to SRAM on the Intel IXP2800 can save about 150 cycles per memory access. Accessing L2 cache on a Core 2 Duo processor is an order of magnitude faster than accessing DRAM. Furthermore, if each core tends to reference its own cache memories (on-chip memory), the shared memory becomes less of a performance bottleneck as more CPU cores are used. In a word, compression is a viable technique to allow the scalability for multi-cores.

Secondly, decompression itself will not become a performance bottleneck. We observe that (1) idle CPU cycles can be utilized for decompression; (2) as more CPU cores are available, it is easier to find more CPU idle cycles on each core since the workload is accordingly reduced. Therefore, decompression will less likely become a new performance bottleneck in multi-cores.

Thirdly, a succinct encoding scheme must be designed to tame the decompression complexity. Normally a high compression ratio comes with high processing power. We need balance the requirements between processing power and space reduction. For example, the decompression code should avoid referencing the external memory to take advantage of data locality.

For classification rules, the IP addresses are represented in prefix and the TCP ports in range. By dividing classification into two stages, range-to-prefix transformation is eliminated. For example, a four-bit range [2 : 14] is transformed into five prefixes 001*, 01**, 10**, 110*, and 1110. If prefix match were used to search all the fields, the number of rules would increase five times, which is contrary to the goal of reducing memory space. Furthermore, previous studies on classifier database characteristics [6] have revealed that 99.9% of the time the number of classifier rules that match a given source-destination prefix pair is no more than 5. By taking advantage of this classification characteristic, **TIC** (**T**wo-stage **I**nterpreting based **C**lassification) is designed to handle prefixes and ranges separately.

In the first-stage the RFC [13] style of reduction tree made from source and destination addresses is searched since RFC is still the fastest classification algorithm. This kind of search can be efficiently implemented by a multithreaded architecture. For instance, multiple outstanding operations can be issued simultaneously from a single thread to overlap CPU execution with memory accesses. In the second stage, an interpreter executes a sequence of ALU instructions to find a matching range. Each instruction is encoded in the CISC (Complex Instruction Set Computer) format in order to reduce the total program size.

By avoiding range-to-prefix transformation and applying instruction-encoding scheme, the TIC algorithm can save up to 97% of memory space compared to RFC, and the resulted classification data structures can be easily fit into the 4MB L2 cache. Furthermore, the instructions are loaded in blocks whose size equals to the cache-line size or internal memory block size. By exploiting such spatial locality, the interpreter can calculate range expressions efficiently. Such interpreting based classification shows good scalability as more cores are available.

To summarize, the goal of this paper is to design and implement a high-performance packet classification algorithm on a multi-core 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 two representative multi-cores: the Intel Core 2 Duo and Intel IXP2800, the same high-performance can be achieved on other similar multi-cores. This paper makes four main contributions:

- It shows that a two-stage interpreting based packet classification can be efficiently implemented on the two distinct yet representative multi-cores. Experiments show that it can reduce space by 97%, its speedup is almost linear, and it can run even faster than 10Gbps on both types of multi-cores.
- It studies and analyzes the performance issues in TIC algorithm design and implementation. We apply the systematical approach to address these issues by incorporating architecture awareness into parallel algorithm design.
- It promotes compression as an effective means to address the speed gap between CPU and memory and between on-chip and off-chip memories in multi-cores. The experimental results indicate that the interpreting based decompression technology is not a performance bottleneck if fetching a cache-line size of data can be efficiently supported in the architecture.
- To the best of our knowledge, TIC is the first across-platform multi-core solution for packet classification that achieves 10Gbps speed. It significantly reduces the space complexity and pre-processing time of the previous fastest RFC algorithm. Our experiences may be applicable to parallelizing other networking applications on the similar multi-cores.

The rest of this paper is organized as follows. Section 2 introduces related work on algorithmic classification schemes. Section 3 formulates the packet classification problem. Section 4 briefly introduces the basic ideas of the RFC algorithm, and presents the TIC algorithm and its design space. Section 5 discusses design decisions made related to the multi-core implementation. Section 6 gives simulation results and performance analysis of TIC. Section 7 presents guidance on

effective network application programming on multi-cores. Finally, section 8 concludes and discusses our future work.

2. Related Work

Prior work on classification algorithms have been reported in [6][7][12][13][14][16][17]. Below we mainly compare algorithmic classification schemes.

Bit vector linear search algorithms [7][16] treat the classification problem as an n-dimensional matching problem and search each dimension separately. When a match is found in one 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) [12] 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 [14] 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.

Trie-based algorithms, such as grid-of-tries [17], build hierarchical radix tree structures. 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 one- or two-dimensional searches, however, their search time and memory requirements increase significantly with the number of search dimensions. For a d-dimensional classifier, the worst-case search time is W^{d-1} and the storage requirement is $O(dWN)$, where N is the number of rules in the classifier and W is the length of IP address. The Extended Grid-of-Tries with Path Compression (EGT-PC) algorithm proposed in [6] describes a two-stage approach: first determine the matched source-destination prefix pair via Grid-of-Tries with Path Compression, and then linearly search a list of candidate rules that match the prefix pair. If k-bit expansion is used, the worst-case search time is $(H + 2)*W/k + L$, where H is the maximum length of the tries, and L is the number of candidate rules.

RFC algorithm [13], which is a generalization of cross-producting [17], is so far the fastest classification algorithm in terms of the worst-case performance. Bitmap is a compress technique widely used in networking. Bitmap has been used in IPv4 forwarding [10][18], IPv6 forwarding [19], and packet classification [5]. We designed the bitmap-RFC [5] classification algorithm that reduces memory space significantly by compressing the cross-product tables. However, the bitmap-RFC relies on an NPU bit-manipulation instruction (POP-COUNT) to achieve high-performance. The TIC algorithm performs compression by using the CISC instruction encoding to save memory space. This interpreting scheme can be efficiently implemented on any multi-core processor while achieving better space reduction than bitmap-RFC.

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. The following IP header fields (5-tuple) are generally used: the source and destination addresses, source and destination ports, and protocol type. 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 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.

Since the speed of a packet classification algorithm is dominated by memory accesses in a single-core, previous studies focus on using the number of memory access to measure the classification speed. For the multi-core architectures, the memory hierarchy such as two levels of cache memories and on-chip and off-chip SRAM memories make such measurement inaccurate. In addition, other optimizations can be used to address the latency issue. Data locality can be exploited to reduce the memory latency, and thread-level parallelism (TLP) to hide memory latency. Therefore, the throughput should be used as a performance metric to measure the classification speed. In this paper, we investigate a way of trading the CPU idle cycles for reducing the memory accesses. We exploit both locality and TLP to reduce and tolerate the memory latency. The goal is to design a scalable classification algorithm that can run on a wide range of multi-cores at 10Gbps speed or higher.

4. Developing TIC Algorithm

4.1 RFC Reduction Tree

Reduction tree is the most important data structure in RFC and it enables RFC to be the fastest classification algorithm. 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.

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

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 "$eqID_0 eqID_1 eqID_2$". 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 field, 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 $Chunk_0[2]*3*2+Chunk_1[4]*2+Chunk_2[4]$, 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.

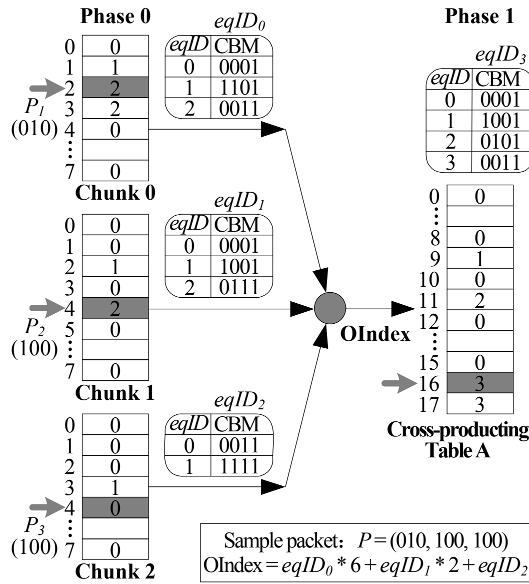


Figure 1. A two-phase RFC reduction tree

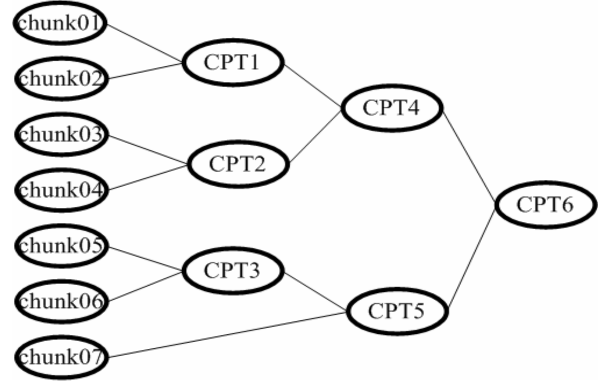


Figure 2. 4-phase RFC reduction tree

A 4-phase RFC reduction tree for 5-tuple IP packet classification is composed in Figure 2. There are seven chunks (corresponding to 16 bits lower/higher src/des IP address, 16 bits src/des port number, 8 bits protocol type respectively) in phase 0, three CPTs in phase 1, two CPTs in phase 2, and one CPT in phase 3. It achieves the fastest classification speed by using 13 memory accesses per packet. Astute readers may notice that the size of CPTs increases non-linearly in later reduction phases. The fast speed of RFC is thus obtained at the cost of memory explosion.

4.2 TIC Algorithm Description

Since the space will explode in RFC when the number of rules becomes large, two-stage classification algorithms [6][11] were proposed to balance classification speed and memory space. In a two-stage classification algorithm, the IP address fields and TCP port numbers are searched separately. The advantages of such separation are as follows:

Firstly, the IP address fields are normally represented in prefix and the TCP port fields in range. Because range-to-prefix transformation increases the number of actual rules, it will increase memory space accordingly. Furthermore, matching a prefix is memory-intensive operation and matching a range is ALU-intensive operation. It will be more efficient to handle them separately, especially on a multi-core where there are idle CPU cycles when more CPU cores are available.

Secondly, previous studies on classifier database characteristics [6] have revealed that 99.9% of the time the number of classifier rules that match a given source-destination prefix pair is no more than 5. Our analysis on the synthetic classifiers generated by ClassBench [4] find that 95.8% of the time the number of matching rules is no more than 10. Therefore, the number of ranges to be matched after the first stage is limited.

Our Two-stage Interpreting based Classification (TIC) algorithm consists of the following two stages:

The first stage is to search a RFC reduction tree composed from the source and destination addresses. As shown the upper part of Figure 2, there are three phases of search with 7 memory accesses. In phase 0, it contains 4 chunks: chunk 0 and 1 searches the low and high 16 bits of source IP address, and chunk 2 and 3 searches the low and high 16 bits of destination IP address respectively. In

phase 1, it contains 2 CPTs made from the results of the 16-bit address match. In phase 2, a list of range expressions is returned.

The second stage is to search a list of range expressions made from the port numbers and protocol fields. Ranges in the list are encoded as a sequence of ALU instructions to further reduce memory space, and an interpreter is implemented to execute those instructions to find a match. The Range Interpreter (RI) is a kind of simple virtual machine. It fetches a block of code from external memory to internal memory (cache), and then it decodes and executes each instruction sequentially. The interpreter exploits data locality by matching the block size to the cache-line size or internal memory block size. By doing so, the external memory accesses are dramatically reduced.

Our analysis on the experimental classifiers shows that 94.9% of lists of candidate ranges can be encoded into one block, whose size is 64 bytes, the same as the cache-line size of an X86 multi-core. Therefore, 94.9% of the time the number of external memory accesses in the second stage is one. On the other hand, since the interpreter only accesses this cache line until all instructions in a block finish execution, it enables highly efficient ALU execution.

4.3 Instruction Encoding

In preprocessing, after the reduction tree of the first stage is constructed, it presents lists of potential matching rules. For the rule example in Table 1, four lists are available, which are (R_4), (R_1, R_4), (R_2, R_4) and (R_3, R_4). Then for each list there is a code block that is formed by encoding the range information in the candidate rules. If one block can not accommodate the whole list, more blocks are needed.

ClassBench [4] gives five classes of port range: WC (wildcard), HI ([1024 : 65535]), LO ([0 : 1023]), AR (arbitrary range), and EM (exact match), and two classes of protocol range: WC and EM. So we have about 50 (5*5*2) operators for the CISC instructions.

There are three instruction formats, with one operand, three operands, and five operands as shown in Figure 3. Since 8 bits are used for encoding the operator and 16 bits for rule ID and the protocol field is 8 bits, the minimum of bytes required for an instruction is 4. For example, operator `EM-WC-WC` means that the protocol field requires exact match, and the source and destination port fields can match any port number since they are a wild character. Therefore, instruction `EM-WC-WC` can be encoded in 4 bytes in which only the protocol field is required in matching. Because an arbitrary range requires two 16-bit numbers to represent, the 8-byte and 12-byte instructions are for these instructions having one or two AR operands. We use 16 bits to store rule ID, which means that the maximum number of rules in the classifiers is 64K.

When port number specification in classifiers is WC, LO or HI, no operand is needed. At least one operand is needed for EM, and two for AR. Table 2 and Table 3 show the distribution of port number specifications in real classifier seeds of ClassBench. For both source and destination port number the proportions of AR are very limited, so instructions of 12-byte appear very infrequently. In fact, the average length of instructions is less than 1.8 long-words.

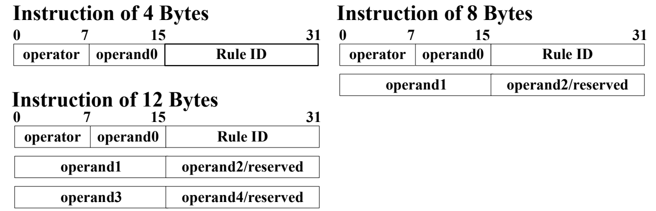


Figure 3. Three instruction formats with 4, 8, and 12 bytes

Table 2. Distribution of source port number

Classifier	WC	HI	LO	EM	AR
seed1	100%	-	-	-	-
seed2	99.90%	-	-	0.10%	-
seed3	99.94%	-	-	0.06%	-
seed4	100%	-	-	-	-
seed5	82.85%	0.35%	-	14.80%	2.00%

Table 3. Distribution of destination port number

Classifier	WC	HI	LO	EM	AR
seed1	30.42%	-	-	57.98%	11.60%
seed2	9.25%	13.96%	-	65.75%	11.04%
seed3	8.56%	12.15%	-	68.08%	11.21%
seed4	30.00%	4.08%	-	60.72%	5.20%
seed5	55.46%	6.52%	-	35.48%	2.53%

In addition, the following instructions are supported:

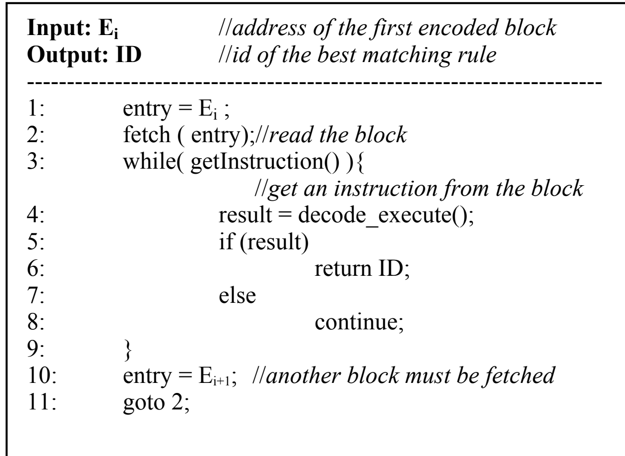
NOP is to avoid handling a partial instruction in a block. An NOP instruction is padded at the end of each instruction block to align the block size to the cache-line size. It simplifies the interpreter execution by eliminating these partial instructions.

HI is to handle well-known TCP port number comparison. TCP port numbers that are less than 1024 are assigned by IETF directly. These two ranges LO ([0 : 1023]), HI ([1024 : 65536]) are widely used in classification rules. Two special instructions are added to denote these two arranges. The size of the two instructions is reduced from 3 to 1 since the two well-known ranges do not need to store into the corresponding instructions.

As Table 2 and Table 3 show that the frequency of the wild character appearance is very high. This suggests to encode this information in the operator field rather than the operand field since the potential number of operators can be as large as 256.

4.4 The Range Interpreter

The pseudocode of the range interpreter is listed in Figure 4. All instruction blocks are stored in external memory, and after the first stage, we get the address of the first code block. In Lines 1-2 the $block_i$ is fetched from external memory to internal memory whose size is equal to the size of cache line. Then (lines 3-4) the current instruction is decoded and executed. If *true* is returned, the best matching rule is found. Thus, it returns the rule ID (lines 5-6); otherwise the search must be continued (lines 7-8). If all instructions in the $block_i$ are executed and still no matching rule is found, then the next $block_{i+1}$ must be fetched (line 10). Please note that the loop (lines 3-9) accesses the current data cache-line only.



```
Input: E_i          //address of the first encoded block
Output: ID          //id of the best matching rule
----------------------------------------------------------------------
1:      entry = E_i ;
2:      fetch ( entry);//read the block
3:      while( getInstruction() ){
                    //get an instruction from the block
4:              result = decode_execute();
5:              if (result)
6:                      return ID;
7:              else
8:                      continue;
9:      }
10:     entry = E_i+1;  //another block must be fetched
11:     goto 2;
```

Figure 4. Pseudocode for the range interpreter

5. Architecture-aware Design and Implementation

We investigate TIC performance in two representative multi-core processors, Intel Core 2 Duo with two levels of cache and Intel IXP2800 without cache. The interesting architectural features of the Core 2 Duo are 4MB L2 cache size and 64B cache line size. We implement TIC in the Pthread library on Linux.

Figure 5 draws the components of the Intel IXP2800, 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-word 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 TIC algorithm in MicroengineC, 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 design issues that have great impacts on algorithm performance.

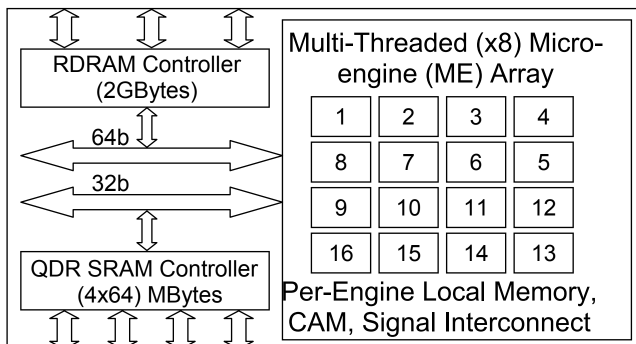


Figure 5. IXP2800 component diagram without I/O interfaces

5.1 Space Reduction

In addition to adopt a two-stage classification scheme to eliminate range-to-prefix transformation to curb the number of rules' growth, we choose the CISC style instruction encoding to further compress the memory space for the list of ranges. It is a well-known fact that CISC instruction encoding produces smaller program size than the RISC encoding. Since space reduction is the first priority, we adopt the CISC encoding to compress the memory space maximally.

Table 4 gives the distribution of instruction sizes for the 5 classifiers used in our experiments, which are generated using the seeds listed in Table 2. On average one third of instructions is 4 bytes. Compared to an 8-byte RISC encoding, the variable-size instruction encoding can save up to 15% of memory.

Table 4. Distribution of Instruction Sizes

Classifier	4 Bytes	8 Bytes	12 Bytes	Average (4 Bytes)
DB1	30.42%	69.58%	-	1.70
DB2	23.13%	76.87%	-	1.77
DB3	20.65%	79.35%	-	1.79
DB4	34.08%	65.92%	-	1.66
DB5	45.53%	54.40%	0.06%	1.55

5.2 Data Locality and Alignment

For a multi-core with two levels of cache memories, we investigate the compression approach to enable data to be stored in L2 cache as much as possible. When fetching a code block, its size must be equal to the size of L2 cache line to help the interpreter exploit the spatial locality. Furthermore, in order to store each encoded block in a cache line, the starting address of code blocks must be properly aligned to avoid cache line false sharing and code block occupying two cache lines.

For the cache-less IXP2800, the single-cycle local memory can be used to store each code block. The maximal size of internal local-memory block is 64 bytes, and we can fetch at most 64 bytes per SRAM read. However, unlike the cache memories which are optimized for a block move, the SRAM operation is optimized for 32-bit operations. Choosing the right size of code block needs careful evaluation. Our study suggests that 32-byte is the right size for the IXP2800.

5.3 Data Partitioning

There are four independent SRAM controllers on the IXP2800 that allow parallel access. To avoid a SRAM controller becoming a bottleneck, we distribute the intermediate tables of the reduction tree and the code blocks uniformly among the four available SRAM memory channels.

5.4 Latency Hiding

On a multi-threaded architecture (such as IXP2800), we hide the latency of memory access by (1) issuing outstanding memory requests whenever possible; (2) overlapping the memory access with the ALU execution [20][21][22]. For example, four memory operations in phase 0 at the first stage can be simultaneously

issued; their memory address calculation can then be overlapped with other memory operations.

On a multi-core architecture (such as Core 2 Duo X86), one CPU core can be used as a helper thread to warm up L2 cache, and then the other main thread can be executed faster if the same cache lines are already fetched. The unified L2 cache on a Core 2 Duo processor can be exploited to reduce the memory access latency.

6. Simulation and Performance Analysis

To overcome the lack of publicly available classifiers, ClassBench [4] was developed for classification benchmarking purpose. The characteristics of the rules produced by ClassBench are close to the real ones', and it is a good benchmark to compare the performance of classification algorithms. We construct the synthetic classifiers from ClassBench. In section 6.1, we introduce the experimental environment. In section 6.2, we show the effectiveness of space reduction. In section 6.3 and 6.4, we demonstrate the scalability of TIC algorithm on Core 2 Duo and IXP processor respectively. In section 6.5, we show the impact of block size on IXP2800.

6.1 Experimental Setup

We use Intel Xeon 5160 Dual-Core running at 3.00GHz with 32KB L1 data cache, 4MB L2 cache, and a 1333MHz system bus. There are two dual cores on the same chip and totally there are four CPU cores. We also use a cycle-accurate IXP2800 simulator to run experiments, and each Microengine runs at 1.2GHz with 8 threads of execution. All 4 SRAM channels are populated with 64MB SRAM each.

We generate packet traces from ClassBench, and use the low locality traces on Xeon 5160 to cancel the locality brought from traces as much as possible.

We generate 10 synthetic classifiers, 5 classifiers having about 2K rules and other 5 having about 4K. By varying the size of these classifiers, we measure the effectiveness of space reduction. Since the classification performance of RFC and TIC on IXP2800 are independent to the size of classifiers and trace locality, we only present the results for the 2K rules.

We use the minimal packets as the worst-case input [15]. 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.

Table 5. Memory size (MBytes) for RFC and TIC

Size	Classifier	#Rules	RFC	TIC	Imp.
2K	*DB1*	1921	2.46	1.52	38%
	DB2	2020	14.86	2.63	82%
	DB3	2008	11.93	2.43	80%
	DB4	1671	2.82	2.07	27%
	DB5	2012	47.31	2.88	94%
4K	*DB6*	3461	2.66	1.53	42%
	DB7	3989	39.17	4.76	88%
	DB8	4009	36.81	5.12	86%
	DB9	2925	2.93	2.04	30%
	DB10	3688	82.52	2.30	97%

6.2 Effective Space Reduction

Table 5 lists the total memory requirements for 4-phase RFC and TIC. For all classifiers, the memory requirements of TIC are smaller than those of RFC. For *DB10*, we have achieved up to 97% of memory size reduction. In addition, the memory size of *DB10* for RFC is larger than 64MB, which is too large to store into any of IXP2800 SRAM channels. Furthermore, the preprocessing time for TIC is under 10 minutes, which are two orders of magnitude smaller than that for RFC. These two features make TIC a good candidate of classification algorithm being put into practical use.

Table 6 shows the memory sizes for two stages (reduction tree and code block) in TIC. Most of the code blocks are within the range of from a few tens of KB to hundreds of KB. With such small memory footprint, the code blocks can be comfortably stored in L2 cache. Furthermore, the combined memory footprint is also less than 4MB when code block is small. This makes TIC scalable when more cores are available.

However, in Table 6 all of the code block sizes are more than 32KB. This indicates that L1 data cache cannot hold these code blocks, and it is difficult to optimize space for L1 cache.

A careful reader may ask: with such a small code block is it worthwhile to do the CISC style encoding to reduce its size? The large code blocks as shown in cases *DB7* and *DB8* justify the necessity to reduce the code block size. In those two cases, the code block sizes are more than 1MB. Without compression, it might be hard to store all entries in the L2 cache. In addition, the size of the code blocks is comparable to that of their counterparts. This is because the classification rules in those two cases have more overlapped source and destination prefixes. This produces the longer lists of the candidate rules after the first stage search. Therefore, the CISC style of encoding is definitely required to tame such memory growth for the second stage of search.

Table 6. TIC memory size (MBytes) for two stages: reduction tree and code block

Classifier	RT	Code	Total
DB1	1.48	0.04	1.52
DB2	2.21	0.42	2.63
DB3	1.99	0.44	2.43
DB4	2.03	0.04	2.07
DB5	2.58	0.30	2.88
DB6	1.47	0.06	1.53
DB7	2.71	2.05	4.76
DB8	4.02	1.10	5.12
DB9	1.98	0.06	2.04
DB10	1.64	0.66	2.30

6.3 Relative Speedups for Core 2 Duo

Table 7 lists the classification speeds of Core 2 Duo for five 2K classifiers in the average cases, and Table 8 for the worst cases. The worst-case classification speed is calculated by the number of packets divided by the longest thread's execution time, and the average case classification speed by the number of packets divided by the mean of thread's execution times.

Table 7. Classifying speeds (Mpps) for RFC and TIC on average cases

		1 T	2 T	3 T	4 T
DB1	***RFC***	15.42	30.78	43.92	56.75
	TIC	12.89	25.31	36.85	47.73
	Imp.	-16.4%	-17.8%	-16.1%	-15.9%
DB2	***RFC***	10.89	21.02	31.27	40.68
	TIC	11.43	22.37	32.37	40.24
	Imp.	4.9%	6.4%	3.5%	-1.1%
DB3	***RFC***	11.47	22.82	33.35	41.48
	TIC	11.72	21.07	33.36	43.28
	Imp.	2.2%	-7.7%	0.0%	4.3%
DB4	***RFC***	14.84	23.53	42.05	54.78
	TIC	13.06	25.95	37.41	44.86
	Imp.	-12.0%	10.3%	-11.0%	-18.1%
DB5	***RFC***	9.05	17.71	24.49	34.87
	TIC	10.64	20.79	26.73	38.88
	Imp.	17.6%	17.4%	9.1%	11.5%
Ave.	***RFC***	12.33	23.17	35.02	45.71
	TIC	11.94	23.09	33.34	42.99
	Imp.	-3.1%	-0.3%	-4.8%	-5.9%

Table 8. Classifying speeds (Mpps) for RFC and TIC on the worst cases

		1 T	2 T	3 T	4 T
DB1	***RFC***	15.42	21.54	24.31	26.61
	TIC	12.89	20.52	25.28	30.02
	Imp.	-16.4%	-4.7%	3.9%	12.8%
DB2	***RFC***	10.89	14.59	17.09	20.49
	TIC	11.43	16.08	18.57	21.13
	Imp.	4.9%	10.2%	8.7%	3.1%
DB3	***RFC***	11.47	15.57	17.96	20.96
	TIC	11.72	16.38	19.73	21.27
	Imp.	2.2%	5.2%	9.9%	1.5%
DB4	***RFC***	14.84	19.08	22.52	24.43
	TIC	13.06	19.43	22.95	24.87
	Imp.	-12.0%	1.8%	1.9%	1.8%
DB5	***RFC***	9.05	12.14	15.19	16.44
	TIC	10.64	14.82	21.63	18.59
	Imp.	17.6%	22.1%	42.4%	13.1%
Ave.	***RFC***	12.33	16.58	19.42	21.78
	TIC	11.94	17.44	21.63	23.18
	Imp.	-3.1%	5.2%	11.4%	6.4%

Table 9. Interpreter characteristics in processing code block

Classifier	Ave. Number of Range Evaluated	Ave. Number of Fetched
DB1	4.28	1.23
DB2	1.42	1.00
DB3	1.64	1.02
DB4	2.31	1.05
DB5	1.17	1.00

We observe the followings:

1. On average three CPU cores are required for TIC to achieve the 10Gbps wire-speed or above. TIC and RFC are comparable in speed since their speed differences are within the single digit range.
2. The performance behavior of the worst-case and the average-case is different. In most cases the TIC performance is better than RFC in the worst-case. The reason is that in the worst-case it mainly measures the helper thread's performance in which it has quite large number of L2 cache misses. In this setting, the interpreter overhead is insignificant. On the other hand, in the average-case, the main thread has a very few L2 cache misses, the interpreter overhead might be noticeable. For example, in Table 8 it shows TIC's worst-case classification speeds are slower than RFC's in 1-thread and 2-thread cases. However, TIC's speeds are faster than RFC in 3-thread and 4-thread cases. Furthermore, when more threads available, TIC's performance is better than RFC's as shown in Table 8 when thread number is 2, 3, and 4. There is one regression of TIC worst-case performance in DB5 when 4 CPU cores are used. We are still investigating the causes of such regression.
3. For the average cases, if the memory space of RFC is smaller than L2 cache size, RFC is better than TIC in terms of classification speeds. For example, for the *DB1* and *DB4* classifiers TIC runs slower than RFC on average. This is because the interpreter overhead becomes noticeable when the memory is not a performance bottleneck. On the other hand, if the memory space of RFC is bigger than L2 cache size, TIC is faster than RFC in terms of classification speed. For *DB2*, *DB3* and *DB5* classifiers, TIC runs faster than RFC in terms of classification speeds. This is because TIC has much less L2 cache misses than RFC which results from that TIC's data structures can be locked into L2 cache after compression.

Table 9 lists the code block behavior in the second stage of interpreting execution. Most of cases, fetching one code-block is enough to find a match, and no more than 2 ranges are searched for a match. For *DB1*, there are on average 4.28 ranges are searched. This explains why the single-thread performance of RFC is better than TIC in *DB1*. However, such advantage will be diminished when more cores are used since the workload is reduced accordingly on each core as shown in Table 8.

6.4 Relative Speedups for IXP2800

Table 10 lists the classification speeds of IXP2800 for five 2K classifiers. On average the classification speed of TIC algorithm reaches 10Gbps speed on 4 Microengines. Only for *DB1*, 10Gbps is ultimately reached when 5 MEs (40 threads) are used.

Surprisingly, the RFC is faster than the TIC even though the TIC has 5 fewer memory accesses than the RFC. Using the cycle-accurate simulator, we found out two causes make TIC run slower.

First, SRAM controller in IXP2800 is optimized for 32-bit access. Even though the total number of SRAM accesses in TIC is less than that of SRAM accesses in RFC, the amount of data read in TIC is larger than that in RFC since at the second stage 32B is read in at once. As shown in Table 11, TIC has 5 fewer memory accesses than RFC, but it accesses 2 more long-words at least. If more than one code block is fetched, the difference of the total words accessed between those two becomes even bigger.

Table 12 compares the SRAM FIFO size for *DB1* in the average and the worst cases. We can see the FIFO size of TIC is bigger than that of RFC for both average and the worst cases. This

indicates that the time of a TIC SRAM operation stays in FIFO is longer than that of a RFC SRAM operation. Thus, it will eventually slow down the TIC's classification speed.

Second, some of 32B read are wasteful and we will analyze its impacts in the following section.

Table 10. Classifying speeds (Mpps) of IXP2800 for RFC & TIC

		1 T	2 T	4 T	8T	16 T	32 T
DB1	***RFC***	1.81	3.59	6.91	11.01	21.06	35.29
	TIC	1.38	2.38	4.76	6.72	12.22	20.49
	Imp.(%)	-23.8	-33.7	-31.1	-38.9	-41.9	-41.9
DB2	***RFC***	1.78	3.59	6.89	10.98	20.61	34.98
	TIC	1.70	3.24	6.25	9.39	18.23	29.98
	Imp.(%)	-4.5	-9.7	-9.3	-14.5	-11.5	-14.3
DB3	***RFC***	1.77	3.57	6.86	10.89	20.43	35.01
	TIC	1.67	3.20	6.19	9.02	17.99	30.07
	Imp.(%)	-5.7	-10.4	-9.8	-17.2	-11.9	-14.1
DB4	***RFC***	1.78	3.58	6.86	10.73	20.75	35.11
	TIC	1.59	3.01	5.79	9.08	17.28	26.90
	Imp.(%)	-10.7	-15.9	-15.6	-15.4	-16.7	-23.4
DB5	***RFC***	1.75	3.51	6.84	10.90	20.59	35.03
	TIC	1.71	3.23	6.24	9.41	18.16	29.58
	Imp.(%)	-2.3	-7.9	-8.8	-13.7	-11.8	-15.6

Table 11. Memory access behavior of RFC vs. TIC

	# Memory Access	# Long-words Accessed
RFC	13	13 LW
TIC	7 + 1 = 8	7 + 8 = 15 LW

Table 12. The SRAM read FIFO average and maximum size for *DB1*

		Cha. #0	Cha. #1	Cha. #2	Cha. #3	Ave.
TIC	*Max*	8	8	8	8	8
	Ave	0.71	1.17	2.70	3.55	2.03
RFC	*Max*	8	7	7	7	7.25
	Ave	2.76	1.34	1.29	1.29	1.67

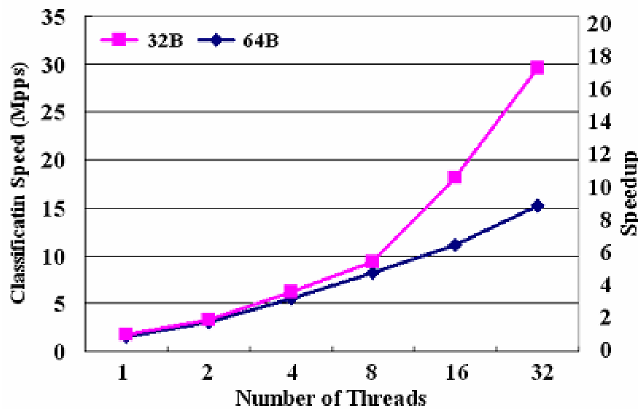


Figure 6. Classifying speeds (Mpps) and speedups for 32B and 64B block size for *DB5*

6.5 Block Size Impact on IXP2800

Figure 6 shows the classifying speeds and speedups of TIC on IXP2800 when the block size is chosen as 32B and 64B respectively for *DB5*. We can see that when the block size is 32B, the classification speed and the speedup is all higher.

There are two reasons: (1) Fetching 64B of code block is wasteful because on average the number of range evaluated is 1.17 (as shown in Table 9); (2) Fetching 64 bytes from SRAM makes the utilization of SRAM channel higher and the size of SRAM read FIFO larger, which causes the latency of SRAM access increasing badly.

As shown in Table 12, on average the average size and the maximal size of SRAM read FIFO is larger in TIC than in RFC. This suggests that a SRAM operation will stay in FIFO longer for TIC than for RFC. Therefore, it takes more time for TIC to access SRAM data than that for RFC. Therefore, choosing the right size of the code block will influence the classification performance.

7. Programming Guidance on Multi-core Architectures

We have presented the implementation and performance analysis of TIC in two representative multi-core processors: Intel Core 2 Duo and Intel IXP2800. Based on our experiences, we provide the following guidelines for creating an efficient network application on a multi-core architecture.

1) Both application and multi-core architectural features must be exploited to design and implement a high efficient networking algorithm.
2) Compress data structures and store them in as high memory hierarchy as possible (e.g., SRAM, cache, on-chip memory) to fill the speed gap between CPU and memory.
3) Proper compression scheme must be chosen to balance the decompression complexity and space reduction, so that decompression will not become a new performance bottleneck.
4) In general, multi-core architecture has many different shared resources. Pay attention to how those shared resources are used because they might become a bottleneck in algorithm implementation.
5) Exploit as many latency hiding techniques as possible to hide memory access latency or reduce memory references.

8. Conclusions and Future Work

This paper proposed a scalable packet classification algorithm TIC that can be efficiently implemented on a multi-core architecture with or without cache. We implemented this algorithm on both Intel IXP2800 network processor and Core 2 Duo X86 architecture. We studied the interaction between the parallel algorithm design and architecture mapping to facilitate efficient algorithm implementation on multi-core architectures. We experimented with an architecture-aware design principle to guarantee the scalability and high-performance of the resulting algorithm. Furthermore, we investigated the main software design issues that have most significant performance impacts on

networking applications. We effectively exploited data structure compression and thread-level parallelism on multi-core architectures to enable an efficient algorithm mapping.

Our performance analysis indicates that we need spend more effort on eliminating various hardware performance bottlenecks, such as the SRAM buses. In addition, how to select an appropriate size of loading block in IXP2800 to make TIC run faster requires more study. We will do more research along these two directions.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments, and Julian Horn (Intel) for proofreading our paper. This work was supported by The Fund for Foreign Scholars in University Research and Teaching Programs under Grant NO. B07033.

References

[1] A. Alameldeen and D. A. Wood. Adaptive Cache Compression for High-performance Processors. *ACM ISCA-31*, Munich, Germany, June 19-23, 2004.

[2] A. Mendelson and J. Mandelblat et. al. CMP Implementation in Systems Based on the Intel Core Duo Processor. *Intel Technology Journal*, Vol. 10(2), 2006.

[3] A.S, Leon and K.W, Tam, et. al. A Power-Efficient High-Throughput 32-Thread SPARC Processor. *IEEE Journal of Solid-State Circuits*, Vol. 42 No. 1, Jan., 2007.

[4] 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.

[5] Duo Liu and Bei Hua and Xianghui Hu and Xinan Tang. High-performance packet classification algorithm for many-core and multithreaded network processor. In *Proceedings of ACM CASES'2006*, Seoul, Korea, pp. 334-344.

[6] 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.

[7] F. Baboescu and G. Varghese. Scalable Packet Classification. In *Proceedings of ACM SIGCOMM*, 2001, pp.199-210.

[8] J. A. Kahle and M. N. Day, et. al. Introduction to the Cell Multiprocessor. *IBM Journal of RES. & DEV*. VOL. 49 NO. 4/5, 2005.

[9] M. Adiletta and Mark Rosenbluth, et. al. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, Vol. 6 (3), August 2002.

[10] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, 1997, pp.3-14.

[11] M. Kounavis et al. Directions in Packet Classification for Network Processors. In *Proceedings of Second Workshop on Network Processors* (NP2), Feb. 2003.

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

[13] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Proceedings of ACM SIGCOMM, Computation Communication Review*, Vol. 29, Sep. 1999, pp.147-160.

[14] S. Singh, F. Baboescu, G. Varghese, and Jia Wang. Packet Classification Using Multidimensional Cutting. In *Proceedings of ACM SIGCOMM'03*, ACM Press, 2003, pp.213-224.

[15] T. Sherwood, G. Varghese and B. Calder. A Pipelined Memory Architecture for High Throughput Network Processors. In *Proceedings of ACM ISCA'03*, 2003.

[16] T. V. Lakshman and D. Stiliadis. High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. In *Proceedings of ACM SIGCOMM98*, Sep. 1998, pp. 191-202.

[17] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel. Fast and Scalable Layer Four Switching. In *Proceedings of ACM SIGCOMM'98*, Sep. 1998, pp. 203-14.

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

[19] Xianghui Hu, Xinan Tang, and Bei Hua. A High-performance IPv6 Forwarding Algorithm for a Multi-core and Multithreaded Network Processor. In *Proceedings of ACM PPoPP'06*, Mar. 2006, pp.168-177.

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

[21] Xinan Tang and Guang R. Gao. 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*, pp. 159-189, 1998.

[22] Xinan Tang, J. Wang, K. Theobald, and Guang R. Gao. Thread partitioning and scheduling based on cost model. In *Proceedings of the ninth annual ACM symposium on Parallel Algorithms and Architectures*, pp. 272-281, 1997.