High-performance IPv6 Forwarding Algorithm for Multi-core and Multithreaded Network Processor

Xianghui Hu
University of Sci. and Tech. of China
Dept. of Computer Sci. and Tech.
Hefei, China, 230027
xhhu@mail.ustc.edu.cn

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

Bei Hua
University of Sci. and Tech. of China
Dept. of Computer Sci. and Tech.
Hefei, China, 230027
bhua@ustc.edu.cn

ABSTRACT

IP forwarding is one of the main bottlenecks in Internet backbone routers, as it requires performing the longest-prefix match at 10Gbps speed or higher. IPv6 forwarding further exacerbates the situation because its search space is quadrupled. We propose a high-performance IPv6 forwarding algorithm TrieC, and implement it efficiently on the Intel IXP2800 network processor (NPU). Programming the multi-core and multithreaded NPU is a daunting task. We study the interaction between the parallel algorithm design and the architecture mapping to facilitate efficient algorithm implementation. We experiment with an architecture-aware design principle to guarantee the high performance of the resulting algorithm.

This paper investigates the main software design issues that have dramatic performance impacts on any NPU based implementation: *memory space reduction*, *instruction selection*, *data allocation*, *task partitioning*, *latency hiding*, and *thread synchronization*. In the paper, we provide insight on how to design an NPU-aware algorithm for high-performance networking applications. Based on the detailed performance analysis of the TrieC algorithm, we provide guidance on developing high-performance networking applications for the multi-core and multithreaded architecture.

Categories and Subject Descriptors C.1.4 [**Processor Architectures**]: Parallel Architectures; C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and Embedded Systems; D.1.3 [**Programming Languages**]: Concurrent Programming – *parallel programming*; D.2.2 [**Software Engineering**]: Design Tools and Techniques;

General Terms Performance, Algorithms, Design, Experimentation

Keywords Network processor, IPv6 forwarding, table lookup, parallel programming, multithreading, pipelining, thread-level parallelism.

This work was supported by the Intel China IXA University Program, the National Natural Science foundation of China under Grant No. 60473068, and the Anhui Province-MOST Co-Key Laboratory of High Performance Computing and Its Application.

PPoPP'06 March 29-31, 2006, New York, NY, USA.

1. INTRODUCTION

Due to the rapid growth of Internet bandwidth and the ever-increasing size of routing tables, IP forwarding (address lookup) has become a big challenge in backbone routers. It will make IP forwarding even more demanding by the inevitable migration from IPv4 32-bit address space to IPv6 128-bit address space.

Traditionally core routers rely on ASIC/FPGA to perform IP forwarding at line-rate speed (10Gbps+) [12][23][24]. As the network processor unit (NPU) emerges as a promising candidate for a networking building block, 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. Up to now, many companies, including Intel[14], Freescale[10], AMCC[3] and Agere[1] have developed their own programmable NPUs.

The advent of the multi-core and multithreaded NPU has given rise to a new paradigm for parallel algorithm design and implementation. Unfortunately, the results of general-purpose multi-processing research are not directly applicable to such system-on-chip (SOC) based multi-core and multithreaded architectures due to their specific processing requirements [17][2]. For example, in order to hide latencies of frequent memory accesses, much more fine-grained interaction is required among tasks on the same core. Furthermore, NPUs need to process packets in real time, which makes the performance budget of the data-path program rather tight. For example, to meet the OC-192 (10Gbps) speed, at most 57 clock cycles are allowed for an Intel 1.4-Ghz IXP2800 to process a minimal IPv4 packet. Under such a stringent constraint, data-path algorithms for multi-core and multithreaded NPUs must consider the following:

- typical parallel programming issues, such as data allocation and task partitioning, to reduce memory latencies;
- micro-architectural factors that impact performance, such as instruction scheduling and selection;
- thread-level parallelism for hiding memory latencies;
- thread synchronization for coordinating potential parallelism among packets.

The savings achieved by these optimization techniques, whether applied manually or by the compiler, may vary from a few cycles to hundreds of cycles. For example, reallocating data from DRAM to SRAM on an Intel IXP2800 can save more than 150 clock cycles per memory access. If two memory accesses are

scheduled in consecutive cycles, the issuing cycle of the second memory access can be hidden completely. We believe that high performance can be achieved through close interaction between algorithm design and architectural mapping, from high-level decisions on data allocation and task partitioning down to low-level micro-architectural decisions on issues such as instruction selection and scheduling. This potential for increased performance motivates the development of architecture-aware networking algorithms that exploit the unique architectural properties of an NPU to achieve high performance. TrieC is one such NPU-aware IPv6 forwarding algorithm specifically designed to exploit the architectural features of the SOC based multi-core and multithreaded systems.

Because the 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 data allocation. For example, moving data from DRAM to SRAM can dramatically reduce memory latencies. In addition, the NPU includes unique architectural features, such as fast bit-manipulation instructions, non-blocking memory access, and hardware supported multithreading. Although these features can be exploited to improve the algorithm performance, making effective use of them is a challenging task for algorithm designers and implementers.

Running IPv6 forwarding at line-rate requires a highly efficient parallel algorithm specifically designed for and implemented on a multi-core and multithreaded NPU architecture. To study potential optimization opportunities, we carefully investigated six software design issues: *space reduction*, *instruction selection*, *data allocation*, *task partitioning*, *latency hiding*, and *thread synchronization*. For each opportunity, we considered the following specific design issues that might be trouble spots in IPv6 forwarding implementation:

- how to compress the routing table effectively while still maintaining fast search and update speed.
- how to use fast bit-manipulation instructions to make the search of compressed routing table faster.
- how to allocate compressed routing tables properly onto the NPU hierarchical memory system to reduce memory access latency while balancing memory access load.
- how to partition work appropriately onto multiple cores and multiple threads to make effective use of on-chip resources.
- how to use multithreading to overlap local computation and memory accesses.
- how to synchronize thread execution on such a multithreaded architecture.

After answering all these questions, we propose a high-performance IPv6 forwarding algorithm TrieC that addresses these issues and runs efficiently on the Intel IXP2800.

To summarize, the goal of this paper is to identify the key issues in designing networking algorithms on an SOC based multi-core and multithreaded NPU, and provide guidance on how effectively to exploit NPU architectural features to address these performance bottlenecks to attain high performance. Although we experimented on the Intel IXP2800, the same high-performance can be achieved on other similar NPUs [1][3][10]. The main contributions of the paper are as follows:

- A scalable IPv6 forwarding algorithm was proposed and efficiently implemented on the IXP2800. Experiments show that its speedup is almost linear and it can run even faster than 10Gbps.
- We carefully studied and analyzed algorithm design and performance issues. We practiced on incorporating architecture awareness into parallel algorithm design.
- We conducted experiments on a cycle-accurate simulator that enables us to provide suggestions to system designers on possible architectural improvements and to compiler writers on compiler optimizations.

To the best of our knowledge, this is the first IPv6 forwarding implementation that achieves 10Gbps speed on an NPU for large routing tables with up to 400,000 entries. Our experiences may be applicable to parallelizing other networking applications on other multi-core and multithreaded NPUs as well. Furthermore, the software design issues studied, such as instruction selection, data allocation, task partitioning, and latency hiding, are essential to make applications run efficiently on a general-purpose multi-core and multithreaded system. The guidelines we provide are helpful for algorithm designers who want to exploit architectural features to develop high-performance applications on these systems.

The rest of this paper is organized as follows. Section 2 introduces related work. Section 3 explains the NPU-aware IPv6 forwarding TrieC algorithm. Section 4 discusses related design issues. Section 5 gives simulation results and performance analysis of Intel IXP2800 implementation. Section 6 presents guidance on effective NPU programming. Finally, section 7 concludes and discusses our future work.

2. RELATED WORK

The most popular data structure for the longest prefix match is a binary trie [21][23]. To reduce memory accesses, various kinds of techniques, such as prefix expansion and multi-bit trie [26], have been proposed. A multi-bit trie expands a set of arbitrary length prefixes to a set of fixed length prefixes to reduce the path length, and thus the memory access times. The fixed prefix lengths are called search strides. However, its worst-case memory requirement is $O(2^k*N*W/k)$, where **k**, **N** and **W** are search stride, number of prefixes, and maximum prefix length, respectively. For example, basic-24-8-DIR [11] is such a hardware implementation of the prefix expansion technique. It requires at most two memory accesses per lookup, but more than 32 Mbytes of memory.

Waldvogel et al employed binary search on a hash table organized by prefix length, which needs $\log_2 W$ memory accesses in the worst case [30]. However, it requires to reconstruct the whole routing table during update in $O(N*\log_2 W)$ time. Multiway range tree [27] reduces both search time and update time to $O(k*\log_k N)$. However, when applied to IPv6 forwarding its memory requirement is as large as $O(k*N*\log_k N)$.

Lampson et al introduced multicolumn search for IPv6 forwarding, which avoids the multiplicative factor of W/M inherent in basic binary search through conducting a binary search in columns of M bits and then moving between columns with pre-

computed information [18]. Its disadvantage is that approximately fifteen memory accesses per lookup are required in the worst case.

The Lulea scheme [8] and Tree Bitmap [9] are closest in spirit to ours in that all use multi-bit tries and bit maps to compress redundant storage in trie nodes. However, both methods try to minimize the memory space at the cost of extra memory accesses. For example, the Lulea algorithm introduces a summary array to pre-compute the number of bits set in the bitmap and thus it needs an extra memory access per trie node. Our algorithm uses a built-in bit-manipulation instruction to calculate the number of bits set, and thus it is much more efficient than Lulea.

Ternary Content Address Memory (TCAM) based schemes [19], reconfigurable fast IP lookup engine [29], Binary Decision Diagrams [24], are hardware-based IP lookup schemes. They achieve high lookup speed at the cost of high power consumption, and complicated prefix update.

There were some efforts in developing high-performance applications for multi-core and multithreaded architectures. Parallelizing non-numeric applications for the *non-blocking* multithreaded architecture was discussed in [25]. Manually parallelizing SPEC2000 for *speculative* multithreaded architecture was reported in [22]. Improving server software performance for *simultaneous* multithreaded architecture was analyzed in [20]. Our paper focuses on designing an architecture-aware IPv6 forwarding algorithm for the multi-core and multithreaded NPUs.

This paper presents an IPv6 forwarding algorithm TrieC, which employs bitmap compression on fixed-level multi-bit trie. This algorithm can conduct high-speed IPv6 lookup while maintaining reasonable memory requirements.

3. BASIC FORWARDING ALGORITHM

IP forwarding algorithms based on the longest prefix match generally adopt a *trie* (tree), in which a prefix is stored along a path from the root to a leaf node. To reduce the path length, and thus memory access times, the prefix expansion technique is applied to visit the expanded routing tables in the fixed strides at the cost of increased table size.

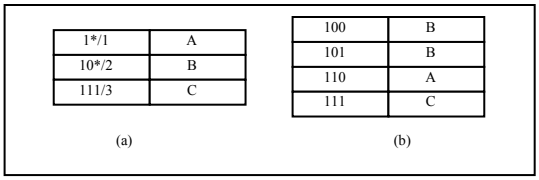

Figure 1. Routing table and its prefix expansion with stride 3

In Figure 1(a), the routing table has three entries (* means don't-care bits), whose prefix length is 1, 2, and 3, respectively. After a 3-bit prefix expansion, the table size increases to 4 (shown in Figure 1(b)), whereas the expanded table can be searched directly using a 3-bit prefix index. Obviously, reducing memory accesses is at the expense of increasing table size. Therefore, the key problem here is how to choose appropriate strides to strike a balance between these two aspects. Gupta and Mckeown used a two-level trie for IPv4 lookup [11], with the most significant 24-bit prefix being the first level, and the remaining 8 bits being the second level.

3.1 IPv6 Forwarding

The length of an IPv6 address is expanded to 128 bits, which makes the straightforward application of prefix expansion impossible due to the explosive memory requirements. Fortunately, the IPv6 routing tables used in core routers have the following characteristics:

- The statistics of existing IPv6 routing tables show that approximately only 5% of the prefixes have a length greater than or equal to 48 bits [15][4].
- Only aggregatable global unicast addresses, those in which the FP field is always set to '001', need to be looked up. Additionally, the lowest 64 bits are allocated to interface ID and should be ignored by core routers [7].

The basic idea of TrieC is to exploit these features by:

- ignoring the highest three bits and the lowest 64 bits
- building a multi-level *compressed* trie tree to cover the prefixes whose lengths are longer than 3 bits and shorter than 49 bits
- searching for remaining prefixes by means of hashing

In view of the resource constraints of NPUs and the high-speed requirements of networking applications, the following design issues should be considered as well:

- Because 2^{n-m} next-hop (routing) information is produced when an m-bit prefix is expanded to an n-bit prefix (m<n), redundancy compression is needed to save memory space.
- Although DRAM is much cheaper today than SRAM, balancing memory speed and memory space is still desirable when designing routing-table data structures.
- Searching compressed Trie trees should not become a new performance bottleneck.
- Multithreading must be used to hide memory latency.

3.2 Modified Compact Prefix Expansion

The preferred IPv6 address notation is x:x:x:x:x:x:x:x, where each 'x' is the hexadecimal value of the corresponding 16 bits in a 128-bit address. The symbol **'::'** represents multiple groups of 16-bit zeros. Similar to the IPv4 prefix notation, an IPv6 address prefix is represented by an "ipv6-address/prefix-length" pair, where the prefix-length is a decimal value specifying the length of a prefix (the leftmost contiguous bits). For example, 12AB:0000:0000:CD30:0000:0000:0000:0000/60 is a legal representation of the 60-bit prefix: 12AB00000000CD3.

The modified compact prefix expansion (MCPE) technique is motivated by the fact that there is much redundant next-hop information in the expanded routing tables after the traditional prefix expansion technique is applied. For example, if IPv6 prefixes (2002:4*::/18,A) and (2002:5*::/20,B) are expanded to 24-bit prefixes, a total of 64(=2^{24-18}) new prefixes are formed as shown in Figure 2(a), where next-hop indices, A appears in two different blocks 48 times, and B appears in one block 16 times.

The basic idea of MCPE is to use a bit vector to compress the continuously identical next-hop index and store those indices only once. Taking Figure 2(b) as an example, the three address blocks, each containing a sequence of identical next-hop index, are compressed to three entries (A, B, A) and stored in the Next-Hop Index Array (NHIA). All 64 address prefixes, whose leftmost 18

bits are the same, are grouped to form an entry in the table, with the leftmost 18 bits acting as the index (Tindex). The lowest 6 bits are used as another index to search a bit vector **BitAtlas** to locate the correct next-hop index in NHIA. Each bit of the 64-bit **BitAtlas** corresponds to an original prefix in the address group, with the least significant bit corresponding to the first prefix of the group. If the next-hop index corresponding to bit ***I*** is the same as that corresponding to bit ***I-1***, bit ***I*** is set to 0; otherwise bit ***I*** is set to 1, indicating that a different next-hop index starts in NHIA. In Figure 2 (b), bit 0 is always set to 1, because it starts a new NHIA entry; bit 16 is set to 1, because its NHIA entry is B, which is different from the previous entry A; bit 32 is set to 1, because its NHIA entry is A, which is different from the previous entry B.

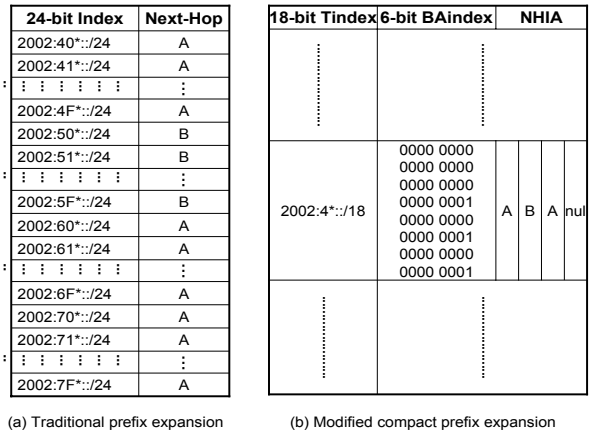

Figure 2. Prefix expansion vs. MCPE

Assume address 2002:6A*:: is searched. Firstly, the highest 18 bits are used as **Tindex** to locate the MCPE entry *2002:4*::/18*. Secondly, the lowest 6 bits '101010'(42) are used as **BAindex** to locate the bit position in **BitAtlas**. As a total of 3 bits are set from bit 0 to bit 42, the third element 'A' in NHIA is the lookup result.

Therefore, the key to accelerating the search of such a compressed tree is to efficiently compute the number of bits set. NPUs with hardware support to such computation are desirable, as they do not create a new performance bottleneck. The Intel IXP2800 has a built-in bit-manipulation instruction, **POP_COUNT**, which can calculate the number of bits set in a 32-bit register in three clock cycles. Therefore, MCPE compression technique can be efficiently implemented on the IXP2800.

The example TrieC table in Figure 2 (b) is called TrieC18/6. Similarly, Trie**C*m/n*** is designed to represent 2^{m+n} compressed (***m***+***n***)-bit prefixes. By using the MCPE technique, the TrieC tree could greatly eliminate redundancy while preserving the high-speed indexing ability of traditional prefix expansion technique.

3.3 Data Structures

The stride series we use is 24-8-8-8-16, and the corresponding trie nodes are three tables: **TrieC15/6** table (ignoring the format prefix field '001'), **TrieC4/4** table, and **Hash16** table. The TrieC15/6 table forms the first level of a tree that stores all the prefixes whose lengths fall into [1:24]. The TrieC4/4 tables form the second to the fourth level of the tree, where prefix lengths belong to [25:32], [33:40], and [41:48], respectively. The **Hash16** table stores all the prefixes whose lengths belong to [49:64].

The next-hop index, which stores the lookup result, is shown in Figure 3(a). Each NHI entry is 2-bytes long. If the most significant bit is set to 0, NHI[14:6] stores the next-hop ID and NHI[5:0] stores the original prefix length. Otherwise, NHI[14:0] contains a pointer to the next level TrieC. The original prefix length is kept for supporting incremental prefix updates [13].

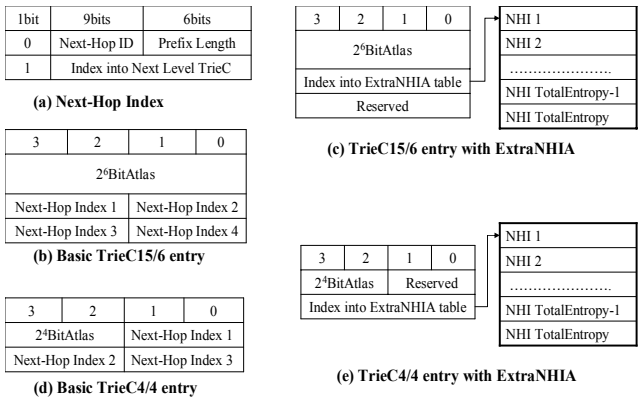

Figure 3. NHI and tables used in the TrieC algorithm

The TrieC15/6 table contains 2^{15} entries named TrieC15/6_entry. Each entry is 16-bytes long, and belongs to one of the two types: ***Basic*** and ***ExtraNHIA***. Basic entry supports up to four NHIs (Figure 3 (b)), and ExtraNHIA allows more NHIs (Figure 3 (c)). For each entry:

1. TrieC15/6_entry[127:64]: stores the 64-bit vector **BitAtlas**. Two terms are used to describe the bit vector. **TotalEntropy** counts the number of bits set in **BitAtlas,** and thus represents the size of NHIA or ExtraNHIA. **PositionEntropy** counts the number of bits set from bit 0 up to a particular bit position in **BitAtlas**.
2. TrieC15/6_entry[63:0]: stores up to 4 NHIs or a pointer to an ExtraNHIA. If **TotalEntropy** is not greater than 4, TrieC15/6_entry[63:0] stores NHI1, NHI2, NHI3 and NHI4 orderly. Otherwise, TrieC15/6_entry[63:32] stores a 32-bit pointer that points to an ExtraNHIA

As in the case of TrieC15/6, each TrieC4/4 table contains 2^4 entries and each entry is 8-bytes long. The structures of the TrieC4/4 entries ***Basic*** and ***ExtraNHIA*** are shown in Figure 3(d) and 3(e), respectively. The fourth level of NHI in the TrieC tree is interpreted slightly differently. If the flag bit is set to 1, TrieC must search the **Hash16** table. The Hash16 table uses a cyclic redundancy check (CRC) as its hash function, which is known as a semi-perfect hash function [16]. The structure of a **Hash16** entry is a (prefix, next-hopID, pointer) triple.

3.4 IPv6 Forwarding Algorithm

Figure 4 displays the pseudo code of an MCPE-based TrieC search algorithm. We will use an example to show how to search these TrieC tables.

Assume that the following routes are already in the TrieC tables: (2002:4C60::/18, A) and (2002:4C6F::/28, B). The first

route requires a TrieC15/6 entry that corresponds to the 24-bit prefixes from 2002:40*::/24 to 2002:7F*::/24. The second route further requires a TrieC4/4 entry on the second level because its length is 28 bits.

```
IPv6_Lookup_TrieC (IN DstIP, OUT Next-HopID) {
1.    Current_Block = TrieC15_6;
2.    Tindex  = DstIP [124:110];
3.    Bit_Vec  = GetBitVec (Current_Block, Tindex);
4.    BAindex = DstIP [109:104];
5.    NHI = GetNHI(Bit_vec, BAindex);
6.    if (NHI.flag = 0) return NHI.Next-HopID;
7.    else { // search TrieC4/4 tables, base[i] is base of (i+1)th-level TrieC tree
8.        Current_Block = TrieC4/4 at Base[0]+NHI[14:0];
9.        for (i=1;i<=3;i++) {
10.           Tindex  = DstIP[103-8*(i-1):100-8*(i-1)];
11.           Bit_vec = GetBitVec (Current_Block, Tindex);
12.           BAindex= DstIP[99-8*(i-1):96-8*(i-1)];
13.           NHI = GetNHI (Bit_Vec, BAindex);
14.           If (NHI.flag = 0) return NHI.Next-HopID;
15.           else {
16.              if (i!=3) Current_Block=TrieC4/4 at Base[i]+NHI [14:0]<<4;
17.              else break;  //search longer prefix in Hash16
18.           }
19.       }
20.       if (Hash (DstIP [79:64])) return Next-HopID;
21.       else return Default-Next-HopID;
22.   }
}// IPv6_Lookup_TrieC
```

Figure 4. Pseudo code to search TrieC multi-level table trees

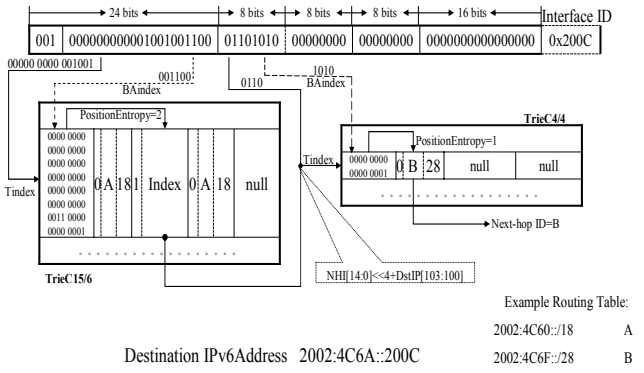

Figure 5. Searching example based on TrieC trees

Consider a search for an IPv6 address, DetIP, 2002:4C6A::200C. Ignoring the leftmost three bits '001' in the format prefix field, ***DstIP***[124:110] (*'0000 0000 0001 001'*) is used to search the **TrieC15/6**, and the TrieC15/6_entry located at position 9 is returned (lines 2-3 in Figure 4). Then ***DstIP***[109:104] (*'001100'*) is used to determine bit position (12) in **BitAtlas**, and the total bits set from bit 0 to bit 12 (**PositionEntropy**) is calculated. Because **PositionEntropy** is 2, the second basic entry is retrieved (lines 4-5). As the flag bit of the NHI entry is 1, the second level TrieC4/4 needs to be further searched (lines 9-22).

Because the base address of the second level of the TrieC tree is NHI[14:0]<<4, NHI[14:0]<<4+DstIP[103:100] is calculated as **Tindex,** and DstIP[99:96] is used as **BAindex,** respectively, to search the second level of the TrieC tree. The **PositionEntropy** of the corresponding TrieC4/4 entry is 1, indicating that the first NHI entry of table TrieC4/4 needs to be examined. The flag bit of this NHI is 0, indicating that NHI[14:6] stores the corresponding next-hop ID for the destination IPv6 address (line 14), and the lookup finishes successfully. Figure 5 shows how various bits are actually used in each search step.

4. NPU-AWARE FORWARDING ALGO.

Figure 6 draws the components of the Intel IXP2800 [14], 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. The average access latency for SRAM is about 150 cycles, and that for DRAM is about 300 cycles. We implemented TrieC 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 design decisions we have made in the implementation of TrieC algorithm for achieving line-rate speed.

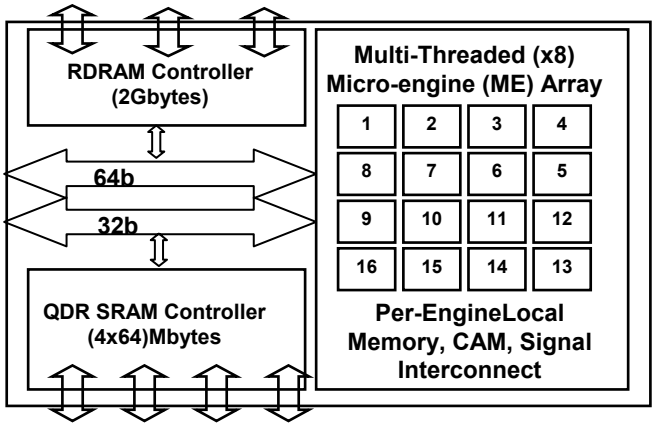

Figure 6. IXP2800 component diagram without I/O Interfaces

4.1 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, to store routing 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 large as that of SRAM. Therefore, tremendous performance gain could be achieved if routing tables could be stored in SRAM. The first decision we made is to compress the routing tables as much as possible to fit them into SRAM. MCPE is such an enabling technique that allows IPv6 routing tables to be stored in SRAM so that the memory access latency will be dramatically reduced.

4.2 Instruction Selection

Searching compressed routing tables requires computing **TotalEntropy** and **PositionEntropy**. These are time-consuming tasks 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 **TotalEntropy** and **PositionEntropy** will become a new performance bottleneck in MCPE-based forwarding algorithm.

With **POP_COUNT** (3-cycle IXP2800 instruction), the number of instructions used to compute **TotalEntropy** and **PositionEntropy** is reduced by more than 97% compared with other RISC/CISC implementations. This is essential for the TrieC algorithm to achieve the line rate.

Another useful bit-manipulation instruction is **BR_BSET** that jumps to a new location if the bit being examined is set. In the TrieC algorithm, the flag bit in an NHI entry is often checked, and then a conditional branch is followed. Because it normally takes three instructions (SHIFT, AND, JUMP) to perform such checking and branching operations in a RISC machine, **BR_BSET** can save two cycles for every checking-and-branching operation.

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.

The last level of the TrieC tree is the **Hash16** table. There are two ways of computing hash in the Intel IXP2800, a centralized hash coprocessor shared by all MEs, and a CRC unit on each ME. We chose CRC to compute the hash function because:

- it provides good hash result for address lookup [16];
- the CRC unit runs much faster than the centralized hash coprocessor does and it takes only 5 clock cycles to compute a 32-bit CRC;
- the CRC unit can run in parallel on each ME. In general, an operation that can run in parallel in each ME is a preferred implementation means for an architecture-aware algorithm.

4.3 Data Allocation

The Intel IXP2800, like other NPUs, has a complex memory hierarchy that comprises single-cycle *local memory*, *scratchpad*, *SRAM*, and *DRAM*. For TrieC implementation, whether SRAM or DRAM to be used, and where and how to distribute the compressed tables greatly affects the ultimate lookup speed.

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

There are four 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 four settings. Experimental results show that the first three settings 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 four SRAM controllers;
- Tables are properly distributed on SRAM and DRAM in a hybrid manner;
- All the tables are distributed on DRAM, and data structures are redesigned to facilitate the burst access.

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 limiting the node sizes of Tries15/6 and TriesC4/4 to less than 64 bytes, memory vectorization optimization can be applied to significantly reduce the number of SRAM accesses.

4.4 Task Partitioning

There are two general ways to partition tasks onto multiple MEs on the Intel IXP2800: *multi-processing* [5] and *context pipelining* [6]. Multi-processing applies two parallelizing techniques. First, multi-threading is applied to a task allocated to one ME. In an Intel IXP2800, a maximum of 8 threads can be used per ME. Secondly, a task can use multiple MEs if needed. For example, if a task needs 2 MEs, a maximum of 16 task threads can run in parallel. Each thread instance runs independently, assuming no other thread instances exist. Such a *run-to-completion* programming model is similar to the sequential one, and it is easy to be implemented. In addition, the workloads are easier to be balanced. However, threads allocated on the same ME must compete for shared resources, including registers, local memory, and command (data) buses. For example, if a task requires more local memory than one ME can support, the context pipelining approach must be used instead.

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

4.5 Latency Hiding

Hiding memory latency is another key to achieving high-performance of TrieC implementation. We hide the memory-access latency by overlapping the memory access with the calculation of bit vector index in the same thread. In Figure 4, operations listed in line 3 and 4 can run in parallel so that the **BAindex** computation is hidden completely by the memory operation GetBitVec(). Similarly, operations in line 11 and 12 can also be overlapped. Compiler based thread scheduling should be able to perform such an optimization automatically [28].

4.6 Packet Ordering

Networking applications normally require packet ordering, which means that packets within the same flow must be sent out in the same order in which they arrived. In other words, only packets belonging to different network flows can run in parallel. We exploit the thread-level parallelism by adopting an *out-of-order*

execution and *in-order-delivery* strategy, in which double signal rings are used to synchronize parallel thread execution.

First, the input packets are assigned to each thread in sequence by implementing a signal ring. Thread 0 sends a signal to thread 1, thread 1 to thread 2, and so on. Thread 7 of one ME sends a signal to thread 0 of its next neighbor ME. Because each thread waits for its turn to receive a packet, each thread starts its execution according to the packet arrival order. On the IXP2800, special instructions can be used to send a signal from one thread to another. They can locate in the same ME or in two different MEs.

Then each thread works independently, without waiting for other threads. In the end, another signal ring is used to enforce the packet exit order in the same way as on the receiving side, i.e., each thread waits for its turn to send its processed packet out. We found out when the length of the signal ring is small, the ring is an efficient way to synchronize thread execution among MEs.

5. SIMULATION AND PERFORMANCE ANALYSIS

Because IPv6 is not yet widely deployed, existing IPv6 tables, which have normally less than 1000 prefixes[15][4], are too small to reflect the future growth of the IPv6 network. We used three different ways to generate nine IPv6 routing tables in the experiment. In order to measure the performance impact of the Intel IXP2800 architecture on the TrieC algorithm, we experimented with the following implementations:

- Using two kinds of bit manipulation instructions to calculate **TotalEntropy** and **PositionEntropy** (see section 5.4)
- Allocating Trie trees onto SRAM, DRAM, and the hybrid of SRAM and DRAM, respectively (see section 5.5)
- Comparing multi-processing vs. context pipelining task allocation model (see section 5.6)
- Overlapping local computation with memory access or conditional branch instructions (see section 5.7)
- With and without enforcing packet order (see section 5.8)

5.1 Experimental Setup

The prefix length distributions of IPv6 routing tables used in the simulation are shown in Table 1.

Table 1. Prefix length distributions

	1-24 bits (%)	25-32 bits (%)	33-40 bits (%)	41-48 bits (%)	49-64 bits (%)	Total (%)
Group A	4.22	69.41	5.52	14.61	6.24	100.00
Group B	0.06	7.40	15.54	71.32	5.68	100.00
Group C	2.14	38.41	10.52	42.97	5.96	100.00

Group A is generated according to the average prefix distributions of CERNET [4], 6Bone, 6Net, and Telstra BGP IPv6 routing tables [15]. It represents the characteristics of existing IPv6 routing tables. Group B is generated according to the non-random IPv6 table generator proposed by M. Wang et al. [31]. It represents the ideal IPv6 routing tables. Group C is the mean of A and B. It represents the future IPv6 tables. In each group, we generated three tables with 200,000, 300,000 and 400,000 entries respectively. All prefix values are generated randomly.

5.2 Compression Effects

The memory requirements of these nine different IPv6 tables are shown in Figure 7. The memory requirement of each table increases along with its increasing table size. Specifically, the memory consumption of table B-400K is approximately 35 Mbytes. It is slightly higher than the 32 Mbytes of the basic-24-8-DIR approach for IPv4 lookup [11]. However, it is significantly less than the estimated memory requirement of a multibit trie, which requires more than 820 Mbytes at the 8-bit stride for 400K IPv6 entries. With such a huge compression rate, the resulted routing tables can be stored in SRAM.

In the worst case, TrieC needs eight memory accesses and one hash operation. As shown in Figure 8, there is no relation between the average memory accesses and the table sizes. The number of average memory accesses depends only on the prefix length distributions of the IPv6 tables. For example, the average memory accesses of group-B are all close to four because the percentages of the 41-48-bit prefixes are all higher than 70%. On average, the number of memory accesses is far less than eight, proving that the percentage of the **ExtraNHIA** nodes is extremely low. In fact, the simulation shows it is only 3.6%.

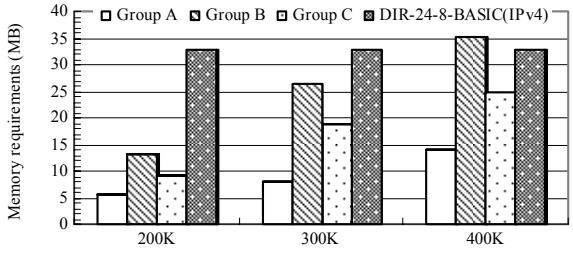

Figure 7. Memory requirements of nine IPv6 tables

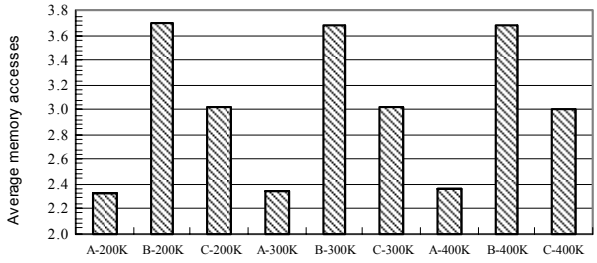

Figure 8. Average memory accesses

5.3 Relative Speedups

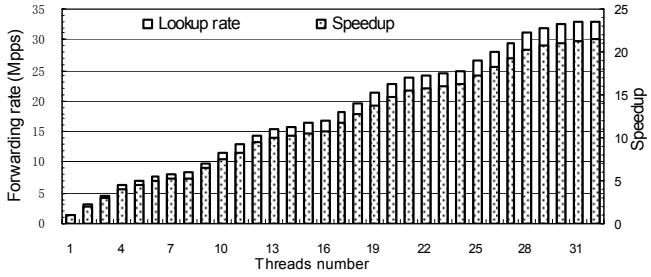

Figure 9. TrieC forwarding rates and relative speedups.

Figure 9 shows the lookup rates and relative speedups using the minimal packet size on the Intel IXP2800. When the minimal IPv6 packet is 69 bytes (9-byte PPP header + 40-byte IPv6 header + 20-byte TCP header), a forwarding rate of **18.16Mpps (Million Packets Per Second)** is required to achieve the OC-192 line rate. The data was collected after all optimizations previously mentioned were applied and the routing tables were stored in four SRAM channels. The speedup is almost linear and it reaches up to **21.45Mpps** for 32 threads. The sub-linear speedup is due to the saturation of the ME command request FIFO. The TrieC is a memory-bound algorithm in which each thread issues eight outstanding memory requests per packet in the worst case. If these memory requests cannot be processed in time, the lookup performance will drop. We found that for each ME, the speedup is linear for up to 4 threads. The slowdown is evident when the number of threads is increased from 5 up to 8. The FIFO fullness ratio, which is defined as the number of elements in the FIFO divided by the FIFO size, can be used to measure whether the memory request FIFO is full or not. It increases from 9% for one thread to 52% for eight threads. That is, in 8-thread mode, a memory request stays six times longer in the FIFO than it does in 1-thread mode. This architectural constraint prevents the TrieC 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 minimal number of threads required to meet the OC-192 line rate. Table 2 shows that on average group A needs only 9 threads, group B 17 threads, and group C 11 threads, respectively.

Table 2. Minimal threads required for supporting line rate

Routing table	Minimum Number of Threads	Forwarding rate (Mpps)	
		Single thread	Multithreads
Group A	9	3.72	23.84
Group B	17	1.81	21.36
Group C	11	2.55	21.32

Considering there are sixteen MEs on the Intel IXP2800, two MEs for IPv6 forwarding use only 1/8 of the entire ME budget. Therefore, TrieC leaves enough room for other networking applications, such as packet classification and traffic management, to meet the line-rate performance.

5.4 Instruction Selection

NPUs that do not have **POP_COUNT** normally support another bit-manipulation instruction, **FFS,** which can find the first bit set in a 32-bit register in one clock cycle. We can compute **TotalEntropy** and **PositionEntropy** with **FFS** by looping through the 32 bits and constantly looking for the next first bit set.

Table 3. Forwarding rates of POP_COUNT vs. FFS

	1 ME	2 MEs	4 MEs	8 MEs
FFS	3.00	5.88	11.99	23.95
POP_COUNT	8.26	16.50	32.14	63.97
Improvement	175%	180%	168%	167%

Table 3 shows the forwarding rates of the worst-case input packets by using two different instructions: **POP_COUNT** and **FFS** respectively. The testing was done for routing table C-400K. The forwarding rate of **POP_COUNT** is much higher than that of **FFS**. On average, the performance improvement of **POP_COUNT** over **FFS** can be as high as 180%.

The lower forwarding rate of FFS is because computational time of **TotalEntropy** depends on the number of bits set in the bit-vectors. 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.

5.5 Memory Impacts

We simulated six data allocation schemes using the worst-case minimal-packet input on the IXP2800. Table 4 shows the simulation results. We found out:

Table 4. Forwarding rates on different data allocations

	1 ME	2 MEs	4 MEs	8 MEs
1-SRAM	8.20	16.43	30.87	33.62
4-SRAM	8.26	16.50	32.14	63.97
DRAM-128	5.59	10.41	12.90	13.04
DRAM-256	4.52	7.29	12.05	12.34
Hybrid-1	5.43	10.79	21.08	21.91
Hybrid-2	4.97	9.46	19.19	21.80

- The 1-SRAM table allocation scheme can support OC-192 line rate with three MEs. However, its speedup is leveled up at four MEs, because the single SRAM channel becomes the bottleneck. The utilization rate of a single SRAM channel is up to 92.64% at 4 MEs and 99.98% at 8 MEs.
- The 4-SRAM configuration obtains almost linear speedup from 1 ME up to 8 MEs. Additionally, the utilization rates of four SRAM channels are all approximately 15% when this configuration meets the OC-192 line rate in the worst case, indicating the potential speedup could be even greater.
- DRAM-128 and DRAM-256 mean that the bit widths of bit-vector are 128 bits and 256 bits, respectively. As mentioned in section 4.3, DRAM on the Intel IXP2800 is optimized for burst accesses of at least 16 bytes. Thus, we redesigned the TrieC algorithm with the stride series 24-24-16 to reduce the number of DRAM memory accesses at the cost of more memory consumption. The simulation shows that both of these two data allocation schemes cannot support the OC-192 line rate in the worst case. The culprit is the DRAM push bus, which is shared by all MEs for reading TrieC trees. This bus has a physical limitation of 2.8Gbytes per second.
- Because the percentage of ExtraNHIA nodes is extremely low and the size of the first level of TrieC tree is fixed, we can reduce the DRAM push bus pressure by allocating them onto SRAM. We experimented on two kinds of hybrid table allocations. The Hybrid-1 configuration stores the first level of the TrieC tree and all ExtraNHIA nodes in SRAM. The Hybrid-2 configuration stores all ExtraNHIA tables in SRAM only. The simulation shows that both of them can support the OC-192 line rate in the worst case with four MEs. However, these two configurations have the disadvantage that their DRAM push bus utilizations still remain high, reaching 87.75% for Hybrid-1 and 96% for Hybrid-2, respectively.

Although the two hybrid allocation schemes both meet the OC-192 line rate, they might not work well in practice because it leaves little DRAM bandwidth for other packet processing applications. DRAM architectural improvement is required before such hybrid schemes can be completely applied in practice. Therefore, the 4-channel SRAM-based allocation scheme is, practically speaking, the best data allocation scheme.

5.6 Task Allocation

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 for the two communication schemes. We divided the whole forwarding task into two pieces according to (1) the algorithm logic; (2) the number of memory accesses required per ME. The partitioning result is as follows:

1. The first and second MEs are for the search of TrieC15/6 table and the 1st-level TrieC4/4 table.
2. The third and fourth MEs are for the search of 2nd-level, 3rd-level TrieC4/4 tables and the Hash16 table.

Because TrieC might end in any stage of context pipelining, 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 5 shows the simulation results using different task allocation policies. It is clear that both multi-processing and context pipelining can support the OC-192 line rate with four MEs on the Intel IXP2800. However multi-processing is preferable for the TrieC algorithm because of the dynamic nature of the workload.

Table 5. Forwarding rate of multiprocessing vs. context pipelining

	1 ME	2 MEs	4 MEs
Multi-processing	8.26	16.50	32.14
Context-pipelining (Scratch ring)	--	--	23.39
Context-pipelining (NN ring)	--	--	28.74

5.7 Latency Hiding

Table 6 reports the performance impact of various latency hiding techniques. The MicroengineC compiler provides only one switch to turn latency hiding optimizations on or off. We reported the combined effects after applying those latency-hiding techniques. The MicroengineC compiler can schedule ALU instructions into the delay slots of a conditional/unconditional branch instruction and a SRAM/DRAM memory instruction.

Table 6. Improvement from latency hiding techniques

	1 ME	2 MEs	4 MEs	8 MEs
Overlapped	8.26	16.50	32.14	63.97
Without overlapped	7.34	14.68	28.83	57.41
Improvement	12.53%	12.40%	11.48%	11.43%

By performing static profiling, we found that twenty ALU instructions were scheduled into delay slots, seventeen in the delay slots of conditional branch instructions and three for memory access. On average, we obtained a performance improvement of approximately 12% by applying the latency hiding techniques.

5.8 Overhead of Enforcing Packet Order

Table 7 shows the overhead of enforcing packet order. Because we use double signal rings, the length of the ring decides the performance impact of sending a signal through the ring. Even though our algorithm uses no more than 8 MEs, we still notice a performance loss of over 13%, indicating a hardware solution might be more profitable to enforce the packet order.

On the other hand, our algorithm can still meet the line-rate even after adding the overhead of enforcing packet order, indicating the scalability of the algorithm implementation. Such good algorithm scalability comes partially from the latency hiding ability provided by the multithreaded architecture and the out-of-order execution strategy adopted.

Table 7. Packet order overhead on TrieC

	1ME	2MEs	4MEs	8MEs
Without packet order	8.26	16.50	32.14	63.97
With packet order	7.26	14.57	28.31	56.49
Overhead	13.75%	13.25%	13.53%	13.24%

6. PROGRAMMING GUIDANCE ON NPU

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

- Compress data structures and store them in SRAM whenever possible to reduce memory access latency.
- Multi-processing is preferred to parallelize network applications rather than context pipelining because the former is insensitive to workload balance. Unless the workload can be statically determined, use a combination of both to help distribute loads among different processing stages fairly.
- In general, the NPU has many different shared resources, such as command and data buses. Pay attention to how you use them because they might become a bottleneck in algorithm implementation.
- The NPU supports powerful bit-manipulation instructions. Select instructions your application needs even without compiler support.
- Use compiler optimizations to schedule ALU instructions to fill the delay slots to hide latency whenever possible.
- The cost of enforcing packet order can not be neglected in practice. A signal ring can be used when the ring is small.

7. CONCLUSIONS AND FUTURE WORK

This paper proposed a high-speed IPv6 forwarding algorithm (TrieC) 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 provided guidlines for creating an efficient network application on

an NPU and effectively exploiting the thread-level parallelism on multi-core and multithreaded architectures.

Our performance analysis indicates that we need spend more effort on eliminating various hardware performance bottlenecks, such as the DRAM push bus. In addition, the heuristics in choosing multi-processing vs. context pipelining require more study. We will do more research along these two directions.

ACKNOWLEDGMENTS

We'd also like to thank the anonymous reviewers for their valuable comments, and Julian Horn, C.J., Lin, and Steve Goodman from Intel, and Prof. Sandhya Dwarkadas from the Univ. of Rochester for helping improve the quality of this paper.

REFERENCES

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

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

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

[4] CERNET BGP View Project, http://bgpview.6test.edu.cn/bgp-view/index.shtml.

[5] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju, "Shangri-La: achieving high performance from compiled network applications while enabling ease of programming", in Proc. of ACM PLDI'05, Chicago, IL, USA, 2005, pp. 224-236.

[6] J. Dai, B. Huang, L. Li, and L. Harrison, "Automatically Partitioning Packet Processing Applications for Pipelined Architectures", in Proc. of ACM PLDI'05, 2005, pp. 237-248.

[7] S. Deering, and R. Hinden, RFC2460, "Internet Protocol, Version 6（IPv6）Specification".

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

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

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

[11] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds", in Proc. of INFOCOM'98, Vol. 3, San Francisco, 1998, pp. 1240-1247.

[12] J. Hasan, and T. N. Vijaykumar, "Dynamic Pipelining: Making IP-lookup Truly Scalable", in Proc. of ACM SIGCOMM'05, Philadelphia, USA, 2005, pp. 205-216.

[13] Xianghui Hu, Bei Hua, and Xinan Tang, "TrieC: A High-Speed IPv6 Lookup with Fast Updates Using Network Processor", in Proc. of the International Conference on Embedded Software and Systems, Xi'an, China, Dec. 2005. pp. 117-128.

[14] Intel, IXP2XXX Product Line of Network Processor, http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm.

[15] IPv6 Report, http://bgp.potaroo.net/index-v6.html.

[16] R. Jain, "A Comparison of Hashing Schemes for Address Lookup in Computer Networks", IEEE Transactions on Communications, 40 (10), Oct. 1992, pp. 1570-1573.

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

[18] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups using Multiway and Multicolumn Search", in Proc. of INFOCOM'98, San Francisco, 1998, pp. 1248-1256.

[19] A.J. McAuley, and P. Francis, "Fast Routing Table Lookup using CAMs", in Proc. of INFOCOM'93, Vol. 3, pp. 1382-1391

[20] L. K. McDowell, S. J. Eggers, and S. D. Gribble, "Improving Server Software Support for Simultaneous Multithreaded Processors", in Proc. of ASPLOS'00, Cambridge, MA, USA, 2000, pp. 245-256.

[21] D. R. Morrison, "PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric", J. ACM, Vol. 15, No. 4, 1968, pp. 514-534.

[22] M. K. Prabhu and K. Olukotun, "Exposing Speculative Thread Parallelism in SPEC2000", in Proc. of ACM PPoPP'05, Chicago, 2005, pp. 142-152.

[23] M. A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms", IEEE Network, Vol. 15, 2001, pp. 8-23.

[24] R. Sangireddy, and A.K. Somani, "High-speed IP Routing with Binary Decision Diagrams based Hardware Address Lookup Engine", IEEE Journal on Selected Areas in Communications, Vol. 21, Issue 4, May 2003, pp. 513-521.

[25] A. Sodan, G. R. Gao, O. Maquelin, J. Schultz, and X. M. Tian, "Experiences with Non-numeric Applications on Multithreaded Architectures", in Proc. of ACM PPoPP'99, Las Vegas, 1999, pp. 124-135.

[26] V. Srinivasan, and G. Varghese, "Fast Address Lookups using Controlled Prefix Expansion", in Proc. of ACM Sigmetrics'98, June 1998, pp. 1-11.

[27] S. Suri, G. Varghese, and P.R. Warkhede, "Multiway Range Trees: Scalable IP Lookup with Fast updates", in Proc. of IEEE GLOBECOM'01, Vol. 3, Nov. 2001, pp. 1610-1614.

[28] Xinan Tang, Guang R. Gao, "Automatically Partitioning Threads for Multithreaded Architectures", in Journal of Parallel Distributed Computing, 58(2): 159-189, 1999

[29] E. Taylor, J. W. Lockwood, T. S. Sproull, J. S. Turner, and D. B. Parlour, "Scalable IP Lookup for Programmable Routers", in Proc. of INFOCOM'02, Vol. 2, pp. 562-571.

[30] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups", in Proc. of ACM SIGCOMM'97, Vol. 27, 1997, pp. 25–36.

[31] M. Wang, S. Deering, T. Hain, and L. Dunn, "Non-random Generator for IPv6 Tables", in Proc. of IEEE Symposium on High Performance Interconnects, 2004, pp. 35-40.