Multi-Terabit IP Lookup Using Parallel Bidirectional Pipelines

Weirong Jiang
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, USA
weirongj@usc.edu

Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, USA
prasanna@usc.edu

ABSTRACT

To meet growing terabit link rates, highly parallel and scalable architectures are needed for IP lookup engines in next generation routers. This paper proposes an SRAM-based multi-pipeline architecture for multi-terabit rate IP lookup. The architecture consists of multiple bidirectional linear pipelines, where each pipeline stores part of a routing table. We address the challenges of realizing such a solution. Two mapping schemes with different granularity are proposed to balance the memory distribution over different pipelines as well as across different stages in each pipeline. Also, IP caching is adopted to facilitate processing multiple packets per clock cycle. Instead of using large reorder buffers and complex logic, a lightweight scheduler and several small output delay queues are developed to preserve the intra-flow packet order. Simulation experiments using real-life data show that the proposed 4-pipeline architecture can store a core routing table with over 200K unique routing prefixes in less than 2 MB of memory, and can achieve a high throughput of up to 18.75 billion packets per second (GPPS), i.e. 6 Tbps for minimum size (40 bytes) packets.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; C.2.6 [**Computer Communication Networks**]: Internetworking—*Routers*

General Terms

Algorithms, Design, Performance

Keywords

IP lookup, Pipeline, Terabit, Bidirectional, SRAM

1. INTRODUCTION

The advent of terabit networks [21] poses a major challenge on the design of next generation IP routers. Some leading industrial vendors are already making efforts to offer multi-terabit core routers [8]. High link rates demand that IP lookup in routers must be performed in hardware. For instance, OC-3072 (160 Gbps) links require a throughput of 1 packet per 2 ns, i.e. 500 million packets per second (MPPS), for a minimum size (40 bytes) packet. Such throughput is impossible using existing software-based solutions [18].

Most hardware-based high-speed IP lookup engines fall into two main categories: TCAM (Ternary Content Addressable Memory)-based and DRAM/ SRAM (dynamic/ static random access memory)-based solutions. Although TCAM-based engines can retrieve IP lookup results in just one clock cycle, their throughput is limited by the relatively low clock rate of TCAMs. TCAMs are expensive and offer little flexibility to adapt to new addressing and routing protocols [7]. As shown in Table 1, SRAMs outperform TCAMs with respect to speed, density and power consumption. However, traditional SRAM-based solutions, most of which can be regarded as some form of tree traversal, need multiple clock cycles to complete a lookup. For example, trie [18], a tree-like data structure representing a collection of prefixes, is widely used in DRAM/SRAM-based solutions. Multiple memory accesses are needed to search a trie to find the longest matched prefix for an IP address.

A number of researchers have explored pipelining to improve significantly the throughput. A simple pipelining approach is to map each trie level onto a pipeline stage with its own memory and processing logic. One IP lookup can be performed every clock cycle. However, this approach results in unbalanced trie node distribution over the pipeline stages. This has been identified as a dominant issue for pipelined architectures [3, 2]. In an unbalanced pipeline, the "fattest" stage, which stores the largest number of trie nodes, becomes a bottleneck. It adversely affects the overall performance of the pipeline in the following aspects. First, more time is needed to access the larger local memory. This leads to a reduction in the global clock rate. Second, a fat stage results in many updates, due to the proportional relationship between the number of updates and the number of trie nodes stored in that stage. Particularly during the update process caused by intensive route insertion, the fattest stage may also result in memory overflow. Furthermore, since it is unclear at hardware design time which stage will be the fattest, we need to allocate memory with the maximum size for each stage. Such an over-provisioning results in memory wastage [2]. To balance the memory distribution across stages, several novel pipeline architectures have been proposed [2, 11, 6]. However, none of them can achieve a

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, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CF'08, May 5–7, 2008, Ischia, Italy.
Copyright 2008 ACM 978-1-60558-077-7/08/05 ...$5.00.

Table 1: Comparison of TCAM and SRAM technologies

	TCAM (18 Mbits chip)	SRAM (18 Mbits chip)
Maximum clock rate (MHz)	266 [16]	400 [5, 19]
Cell size (# of transistors per bit)	16	6
Power consumption (Watts)	$12 \sim 15$ [24]	≈ 0.1 [4]

perfectly balanced memory distribution over stages. Some of them use non-linear structures, which result in throughput degradation, delay variation, and packet blocking during a route update.

Furthermore, the "memory wall" [14] tends to impede the performance improvement of a single pipeline architecture. Thus it becomes necessary to employ multiple pipelines operating in parallel to speed IP lookup. Each pipeline stores part of the routing table so that both power and memory efficiency can be achieved. Similar to the above analysis of how the fattest stage affects the global performance of a pipeline, the fattest pipeline is a performance bottleneck of the multi-pipeline architecture as well. Hence an efficient routing table partitioning and mapping scheme is needed to balance the memory requirement over different pipelines. On the other hand, traffic balancing is needed to achieve multiplicative throughput improvement. Previous work on parallel TCAM-based IP lookup engines uses either a learning algorithm to predict the future behavior of incoming traffic based on its current distribution [24], or IP/prefix caching to utilize the locality of Internet traffic [1]. The former requires periodic reconstruction of the entire routing table, resulting in high overhead of route updates for SRAM-based pipeline solutions. Hence we adopt caching in our architecture.

Due to caching and queuing, packets within a flow[1] may go out of order. This adversely affects some network applications [20]. Hence, expensive reorder buffers and complicated logic are usually needed. The proposed solution preserves the intra-flow packet order without using large reorder buffers.

We propose an SRAM-based multi-pipeline architecture that consists of multiple bidirectional linear pipelines, for high throughput IP lookup. This paper makes the following contributions:

- To the best of our knowledge, this work is among the first discussions of SRAM-based multi-pipeline solutions for multi-terabit IP lookup.
- To balance the memory distribution among different pipelines, a simple but efficient method is proposed for trie partitioning, and an approximation algorithm is used to map subtries to pipelines.
- Within each pipeline, a bidirectional fine-grained node-to-stage mapping scheme is proposed to achieve a perfectly balanced memory allocation over pipeline stages. The memory wastage due to over-provisioning [2] is almost zero.
- IP caching is employed effectively to exploit the Internet traffic locality. A high throughput of nearly 8 packets per clock cycle is obtained in the proposed four-pipeline architecture.
- A lightweight scheduler and several small output delay queues are developed to maintain the intra-flow packet order. Neither a large reorder buffer nor complex reorder logic is needed.
- Our simulation experiments using real-life data demonstrate the SRAM-based pipelined architecture to be a promising solution for next generation IP routers. The proposed 4-pipeline architecture can store a full backbone routing table with over 200K unique prefixes using less than 2MB of memory. It can achieve a high throughput of up to 18.75 billion packets per second (GPPS), i.e. 6 Tbps for minimum size (40 bytes) packets.

The remainder of this paper is organized as follows. Section 2 reviews the background and related work. Section 3 proposes a parallel architecture with multiple bidirectional linear pipelines. The two key problems – memory balancing and traffic balancing – are discussed in Sections 4 and 5, respectively. In Section 6, the results of our experiments are presented and discussed to evaluate the effectiveness of our approaches. Section 7 concludes the paper.

2. BACKGROUND

2.1 Trie-based IP Lookup

IP lookup is the core function of the IP routers, retrieving the next-hop information for each incoming IP packet by matching the destination IP address of the packet with a set of prefixes. The nature of IP lookup is longest prefix matching (LPM). The most common data structure in algorithmic solutions for performing LPM is some form of trie [18]. A trie is a binary tree, where a prefix is represented by a node. The value of the prefix corresponds to the path from the root of the tree to the node representing the prefix. The branching decisions are made based on the consecutive bits in the prefix. A trie is called a uni-bit trie if only one bit is used to make branching decision at a time. The prefix set in Figure 1 (a) corresponds to the uni-bit trie in Figure 1 (b). For example, the prefix "010*" corresponds to the path starting at the root and ending in node P3: first a left-turn (0), then a right-turn (1), and finally a turn to the left (0). Each trie node contains two fields: the represented prefix and the pointer to the child nodes. By using the optimization called *leaf-pushing* [22], each node needs only one field: either the pointer to the next-hop address or the pointer to the child nodes. Figure 1 (c) shows the leaf-pushed uni-bit trie derived from Figure 1 (b).

[1]A *flow* is usually identified by the common fields of IP headers, e.g. typically the five tuple of the source and destination IP addresses, source and destination port numbers and the protocol number [20]. This paper focuses on IP lookup, and thus we define a sequence of packets with the same destination IP address as a *flow*.

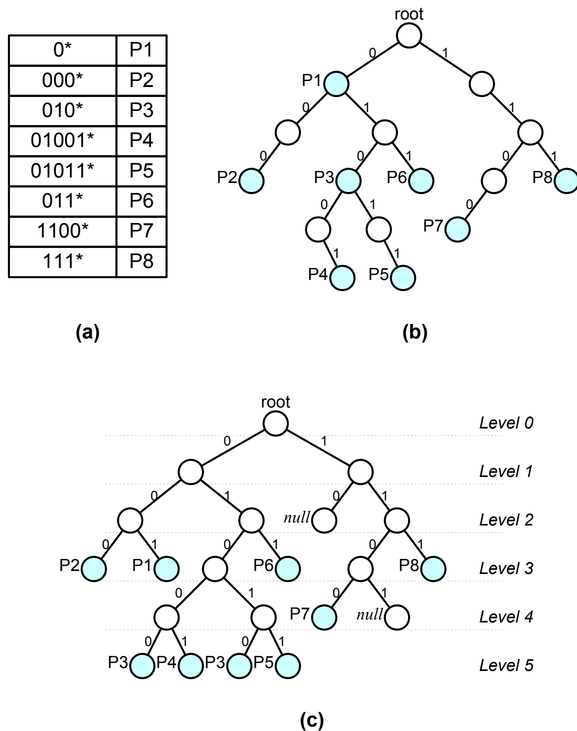


Figure 1: (a) Prefix set; (b) Uni-bit trie; (c) Leaf-pushed uni-bit trie.

Given a leaf-pushed uni-bit trie, IP lookup is performed by traversing the trie according to the bits in the IP address. When a leaf is reached, the prefix associated with the leaf is the longest matched prefix for that IP address. The corresponding next-hop information of that prefix is then retrieved. The time to look up a uni-bit trie is equal to the prefix length. The use of multiple bits in one scan can increase the search speed. Such a trie is called a multi-bit trie. The number of bits scanned at a time is called the *stride*. For simplicity, we consider only the leaf-pushed uni-bit trie in this paper, though our ideas are applicable to other forms of tries.

2.2 Memory-Balanced Pipelines

Pipelining can dramatically improve the throughput of trie-based IP lookup. A straightforward way to pipeline a trie is to assign each trie level to a dedicated stage, so that a packet can be processed every clock cycle. However, as discussed earlier, this simple pipeline scheme results in unbalanced memory distribution, leading to low throughput and inefficient memory allocation.

Basu et al. [3] and Kim et al. [9] both reduce the memory imbalance by using variable strides to minimize the largest trie level. However, even with their schemes, the size of the memory of different stages can have a large variation. As an improvement upon [9], Lu et al. [13] propose a tree-packing heuristic to balance the memory further, but it does not solve the fundamental problem of how to retrieve one node's descendants which are not allocated in the following stage. Furthermore, a variable stride multi-bit trie is difficult for hardware implementation, especially if incremental updating is needed [3].

Baboescu et al. [2] propose a Ring pipeline architecture for tree-based search engines in IP routers. The pipeline stages are configured in a circular, multi-point access pipeline so that the search can be initiated at any stage. A tree is split into many small subtrees of equal size. These subtrees are then mapped to different stages to create a nearly balanced pipeline. Some subtrees must wrap around if their roots are mapped to the last several stages. Any incoming IP packet must look up an index table to find its corresponding subtree's root, which is the starting point of that search. Though all IP packets enter the pipeline from the first stage, their lookup processes may be activated at different stages. All the packets must traverse the pipeline twice to complete the tree traversal. The throughput is thus 0.5 packets per clock cycle.

Kumar et al. [11] extend the circular pipeline with a new architecture called the Circular, Adaptive and Monotonic Pipeline (CAMP). It uses several initial bits (i.e. initial stride) as the hashing index to partition the trie. Using a similar idea but different mapping algorithm than Ring [2], CAMP also creates a nearly balanced pipeline. Unlike the Ring pipeline, CAMP has multiple entry and exit stages. Several queues are employed to manage the access conflicts between packets from current and preceding stages. Since different packets of an input stream may have different entry and exit stages, the ordering of the packet stream is lost when passing through CAMP. Assuming the packets traverse all the stages, when the packet arrival rate exceeds 0.8 packets per clock cycle, some packets may be discarded [11]. In other words, the worst-case throughput is 0.8 packets per clock cycle. Also in CAMP, a queue adds extra delay for each packet, which may result in out-of-order output as well as delay variation.

Due to their non-linear structures, neither the Ring pipeline nor CAMP in the worst case can maintain a throughput of one packet per clock cycle. Also, neither of them supports the non-blocking route update, since the ongoing update may conflict with the preceding or following packets. Our previous work [6] adopts an optimized linear pipeline architecture, named OLP, to achieve a throughput of one packet per clock cycle, while supporting non-blocking route update. By adding *nop*s (no-operations) in the pipeline, OLP offers more freedom in mapping trie nodes to pipeline stages. The trie is partitioned, and all subtries are converted into queues and mapped onto the pipeline from the first stage. However, the first several stages in OLP may not be balanced, since the top levels of a trie have few nodes.

Overall, none of the existing memory-based pipelined solutions can achieve either a perfectly balanced memory distribution across pipeline stages, or a scalable throughput of more than one packet per clock cycle.

2.3 Partition-based Parallel Engines

Little work has been done on parallel IP lookup engines based on SRAM-based pipelines. Most published parallel IP lookup engines are TCAM-based [23, 24]. They partition the full routing table into several blocks, and make the multiple search processes parallel on these blocks. Both power efficiency and throughput improvement can be obtained by such partitioning and parallelization. Some ideas of TCAM-based solutions can be borrowed to develop our solution, but they must be adapted for SRAM-based pipelined architectures.

Two methods are widely used in TCAM-based solutions to partition a routing table [23]: bit-selection and trie-based approaches. In the former, selected bits are used to index different TCAM blocks directly. However, prefix and memory distribution imbalance among the TCAM blocks may be quite high, resulting in poor worst-case performance. The latter scheme splits the trie by carving subtries out of the full trie. This can have a much better worst-case bound. Since those subtries may be on different levels of the trie, different numbers of bits are used to index different subtries. Such a scheme is difficult for SRAM-based solutions, where the index tables are addressable memory with a constant number of address bits.

Traffic balancing is another difficult problem for parallel IP lookup engines. Many solutions based on TCAMs have been proposed, including learning-based block rearrangement [24] and IP/prefix caching [1]. The former requires periodic reconstruction of the entire routing table, which is impractical for SRAM-based pipeline solutions due to the high overhead of updating. On the other hand, because of Internet traffic locality, IP/prefix caching is effective for speeding up the lookup throughput [7]. However, due to caching and queuing, packets within a flow may go out of order. This adversely affects some network applications [20]. As a result, large reorder buffers and logic are usually needed, which are expensive and complicated.

3. ARCHITECTURE OVERVIEW

First, we give the following definitions and notations.

Definition 1. *The* ***input width*** *is the maximum number of input packets per clock cycle, denoted W.*

Definition 2. *The* ***pipeline scale*** *is the number of pipelines, denoted P. Usually $W = 2P$.*

Definition 3. *The* ***pipeline depth*** *is the number of pipeline stages of each pipeline, denoted H.*

Definition 4. *The* ***cache size*** *is the maximum number of IP addresses allowed to be cached, denoted C.*

Definition 5. *The* ***queue size*** *is the maximum number of packets allowed to be stored in a queue, denoted Q.*

The proposed architecture consists of P of SRAM-based bidirectional linear pipelines with the same depth. Figure 2 shows an example of the architecture with $P = 4, W = 8$.

Each pipeline has two entrances and stores part of the full routing table. To partition the routing table, we construct the prefixes of the routing table as a leaf-pushed uni-bit trie and partition the trie into many disjoint subtries. Those subtries are then mapped onto different pipelines while keeping the memory requirement over different pipelines balanced. Within each pipeline, a bidirectional fine-grained node-to-stage mapping is employed to balance the trie node distribution across the pipeline stages. The details of trie partitioning, subtrie-to-pipeline and node-to-stage mapping are discussed in Section 4.

The architecture can be divided into two parts: Front End and Back End. The Front End receives packets and dispatches the packets to different pipelines. The Back End processes the packets and outputs the retrieved next-hop information.

3.1 Front End

At most W packets can be inputted in one clock cycle. The destination IP address of each packet is used to access the cache and the destination index table (DIT) simultaneously. There are W copies of caches and DITs. The cache stores the most recently searched IP addresses and their next-hop information. The DIT stores the relationship between the subtries and the pipeline entrances. As we will see later, both the subtrie-to-pipeline and the node-to-stage mapping schemes determine the pipeline entrance for each subtrie.

After a packet obtains the outputs from the cache and the DIT, the scheduler determines which pipeline entrance the packet is routed to. If the packet gets a cache miss, it is directed to the pipeline entrance for the subtrie to which this packet belongs. Since it is possible for multiple packets to be directed to the same pipeline entrance, each entrance needs a multi-port queue to tolerate the access conflict. If the packet gets a cache hit, it is directed to the queue with fewest packets. Then the packet goes through the queue and the pipeline without any operation, since it already retrieves the next-hop information from the cache.

3.2 Back End

Each pipeline is configured as a dual-entrance dual-exit bidirectional linear pipeline. After a packet enters the pipeline, with the bits of its destination IP address being scanned, the packet traverses the corresponding subtrie by going through all the stages of the pipeline to which the subtrie is mapped, in the direction determined by how the subtrie is mapped. At each pipeline stage, the memory has dual Read/Write ports so that packets from both directions can access the memory simultaneously.

When the packet exits the pipeline, it may be delayed to be output so that the intra-flow packet order is preserved. The number of clock cycles to be delayed for each packet is determined by the scheduler when the packet is at the Front End. To implement the variable delay for each packet, we develop several output delay queues, which are built as single-entrance multi-exit pipelines. Each exit of a pipeline has a dedicated output delay queue. If the architecture has W pipeline entrances, it has W pipeline exits and W output delay queues. The details of the intra-flow packet order preservation are discussed in Section 5.

3.3 Route Updates

We update the memory in the pipeline by inserting *write bubbles* [3]. The new content of the memory is computed offline. When an update is initiated, a write bubble is inserted into the pipeline. The pipeline entrance of a write bubble is the pipeline entrance for the subtrie that the write bubble is going to update. Each write bubble is assigned an ID. There is one write bubble table in each stage. The table stores the update information associated with the write bubble ID. When a write bubble arrives at the stage prior to the stage to be updated, the write bubble uses its ID to look up the write bubble table. Then the bubble retrieves (1) the memory address to be updated in the next stage, (2) the new content for that memory location, and (3) a write enable bit. If the write enable bit is set, the write bubble will use the new content to update the memory location in the next stage. For one route update, we may need to insert multiple write bubbles consecutively.

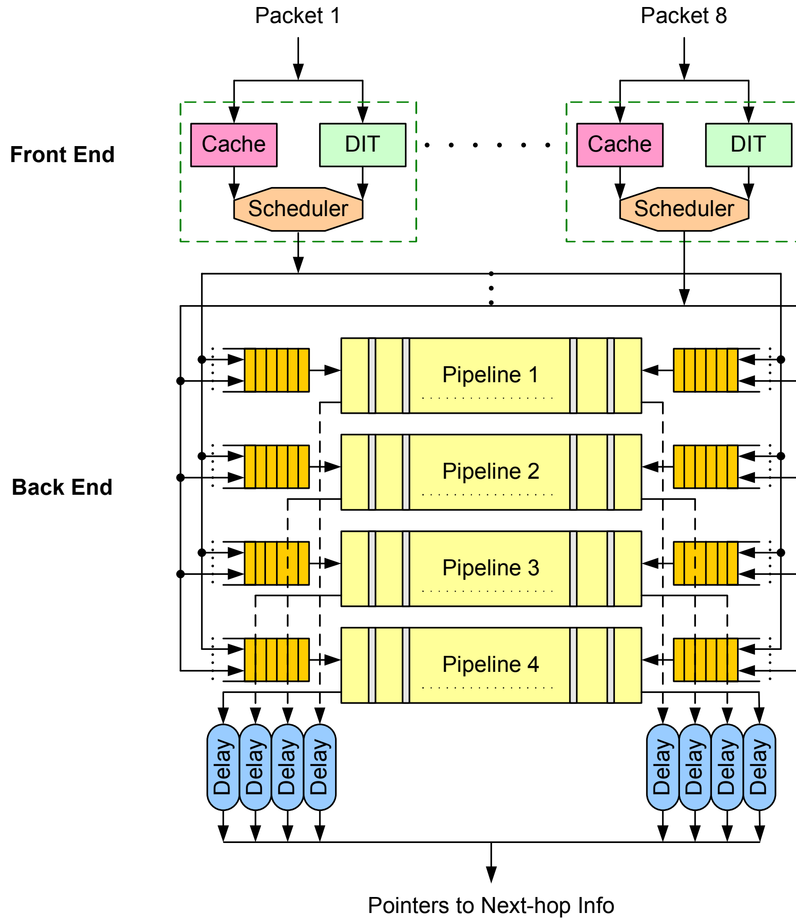


Figure 2: Block diagram of the architecture. ($P = 4, W = 8$)

Since the subtries mapped onto the two directions of each pipeline are disjoint, a write bubble inserted from one direction will not contaminate the memory content for the packet from the other direction. Also, since the pipeline is linear, all packets preceding or following the write bubble can perform their IP lookup while the write bubble performs an update.

4. MEMORY BALANCING

This section studies the problem of memory balancing over the pipelines, as well as across the stages of each pipeline. Three issues are to be addressed.

1. Partitioning the entire routing trie in a simple but efficient way
2. Mapping subtries to different pipelines so that each pipeline has the same number of trie nodes
3. Mapping trie nodes to pipeline stages so that the memory requirement across the stages is balanced

First, we define the following terms.

DEFINITION 6. *The **size** of a trie is the number of trie nodes in the trie.*

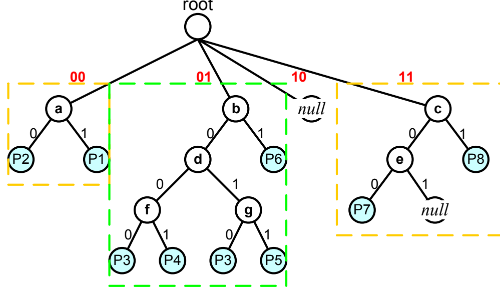
DEFINITION 7. *The **depth** of a trie node is the directed distance from the trie node to the trie root. The depth of a trie refers to the maximum depth of all trie leaves.*

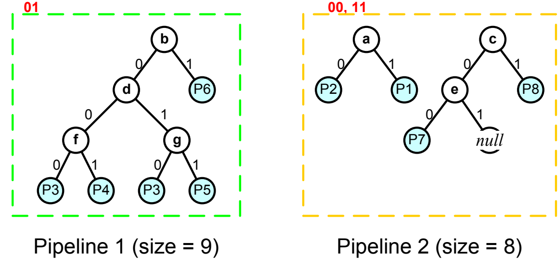
DEFINITION 8. *The **height** of a trie node is the maximum directed distance from the trie node to a leaf node. The height of a trie refers to the height of the trie root. Note that the depth of a trie is equal to its height.*

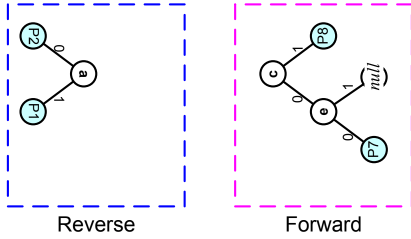
DEFINITION 9. *Two subtries are **disjoint** if they share no prefix.*

4.1 Trie Partitioning

To partition the trie, we adopt a scheme called prefix expansion [22], illustrated in Figure 3. Several initial bits are used as the index to partition the trie into many disjoint subtries. The number of initial bits to be used is called the *initial stride*, denoted I. A larger I can result in more small subtries, which can help balance the memory distribution when mapping subtries to pipelines. However, a large I can result in prefix duplication, where a prefix may be copied to multiple subtries. For example, if we use $I = 4$ to expand the prefixes in Figure 1 (a), the prefix P3 whose length is 3 will be copied to two subtries. One subtrie with the initial bits of "0100" has the prefixes P3 and P4, and the other with "0101" has the prefixes P3 and P5. Prefix duplication results in memory inefficiency and may increase the update cost. If two subtries containing a same prefix are mapped onto two pipelines, a route update related to that prefix must update both pipelines.







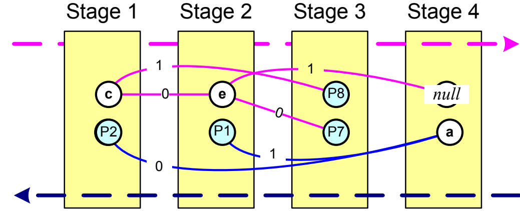


Figure 3: Mapping the trie shown in Figure 1 onto two 4-stage pipelines.

We study the prefix length distribution based on four representative routing tables collected from [17]: JPIX, MAE-WEST, MIX and PAIX. As shown in Table 2, few prefixes are shorter than 16. Hence, using an I of less than 16 should not result in much duplication of prefixes. In the following sections, we pick $I = 12$ as default.

4.2 Subtrie-to-Pipeline Mapping

Our partitioning scheme may result in many subtries of various sizes. For example, using $I = 12$ to partition the trie corresponding to the routing table PAIX, the largest subtrie has 5319 nodes, while the smallest subtrie has only one node.

4.2.1 Problem Formulation

The problem now is to map the subtries to the pipelines so that all pipelines have an equal number of trie nodes:

$$\min \max_{i=1,2,\cdots,P} size(S_i) \tag{1}$$

with the constraint

$$\bigcup_{i=1,2,\cdots,P} S_i = \bigcup_{j=1,2,\cdots,K} T_j \tag{2}$$

where S_i denotes the set of subtries contained in the ith pipeline, $i = 1, 2, \cdots, P$; K the number of subtries, T_i the ith subtrie, $i = 1, 2, \cdots, K$, and $size(.)$ the number of trie nodes of a set of subtries.

4.2.2 Mapping Algorithm

The above optimization problem is *NP-hard*. This can be shown by a reduction from the partitioning problem [10]. We propose a polynomial-time approximation algorithm (Algorithm 1). The complexity of this algorithm is $O(KP)$, where K denotes the number of subtries and P the number of pipelines. According to [10], in the worst-case, the resulting largest pipeline may have 1.5 times the number of nodes as the optimal mapping. Figure 3 illustrates an example of mapping 3 subtries to 2 pipelines.

Algorithm 1 Subtrie-to-pipeline mapping

Input: K subtries: T_i, $i = 1, 2, \cdots, K$.
Input: P empty pipelines.
Output: P pipelines, each of which contains a set of subtries S_i, $i = 1, 2, \cdots, P$.
1: Set $S_i = \phi$ for all pipelines, $i = 1, 2, \cdots, P$.
2: Sort $\{T_i\}$ in the decreasing order of $size(T_i)$, $i = 1, 2, \cdots, K$.
3: Assume that $size(T_1) \geq size(T_2) \geq \cdots \geq size(T_K)$.
4: **for** $i = 1$ to K **do**
5: Find S_m so that $size(S_m) = \min_{j=1,2,\cdots,P} S_j$.
6: Assign T_i to the m th pipeline: $S_m \leftarrow S_m \cup T_i$.
7: **end for**

4.2.3 Experimental Results

To verify the effectiveness of the above algorithm, we used the algorithm on the four routing tables given in Table 2. In these experiments, we set $P = 4$. The resulting size of each pipeline is shown in Figure 4. The size of each pipeline was normalized by:

$$size(S_i)_{normalized} = \frac{size(S_i)}{\min_{j=1}^{P} size(S_j)}, \quad i = 1, 2, \cdots, P. \tag{3}$$

Table 2: Representative Routing Tables

Routing table	Location	Date	# of prefixes	# of prefixes with length < 16
JPIX (rrc06)	Otemachi, Japan	20071130	239332	1926 (0.80%)
MAE-WEST (rrc08)	San Jose, USA	20040901	83556	495 (0.59%)
MIX (rrc10)	Milan, Italy	20071130	236991	1939 (0.82%)
PAIX (rrc14)	Palo Alto, USA	20071130	243731	1949 (0.80%)

where S_i denotes the set of subtries contained by the ith pipeline, $i = 1, 2, \cdots, P$.

According to Figure 4, our algorithm resulted in balanced memory distribution among 4 pipelines for all four routing tables.

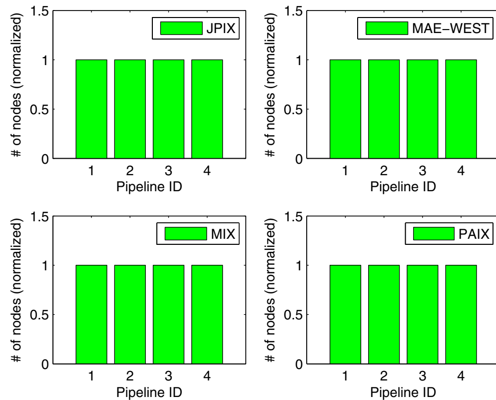


Figure 4: Normalized node distribution over 4 pipelines.

4.3 Node-to-Stage Mapping

We now have a set of subtries for each pipeline. Within each pipeline, the trie nodes should be mapped to the stages while keeping the memory requirement across the stages balanced. Also, each pipeline should be linear in either of the two directions.

4.3.1 Problem Formulation

Consider K subtries mapped to a pipeline. The problem is formulated as:

$$\min \max_{i=1,2,\cdots,H} M_i \tag{4}$$

with the constraint (5) and *Constraint 1*

$$\sum_{i=1}^{H} M_i = \sum_{j=1}^{K} size(T_j) \tag{5}$$

Constraint 1: If node A is an ancestor of node B in a trie, then A must be mapped to a stage preceding the stage to which B is mapped.

In the above formulation, M_i denotes the number of nodes mapped to the ith stage, $i = 1, 2, \cdots, H$; T_i the ith subtrie, $i = 1, 2, \cdots, K$; and $size(.)$ the size of a subtrie, i.e. the number of trie nodes in the subtrie. Note that, due to *Constraint 1*, the pipeline depth must be larger than the height of any subtrie, to guarantee all subtries are mapped to the pipeline.

4.3.2 Motivation of Bidirectional Fine-Grained Mapping

State-of-the-art pipelined solutions [2, 11, 6] cannot achieve perfectly balanced memory distribution, due to several constraints placed during mapping: (1) They require trie nodes on the same level be mapped onto the same stage. (2) The mapping scheme is uni-directional: the subtries partitioned from the original trie must be mapped in the same direction (either from the root or from the leaves). Actually, both constraints are unnecessary. The only constraint we must obey during mapping is *Constraint 1*.

We propose a bidirectional fine-grained mapping scheme, as shown in Figure 3. The main ideas are to allow (1) two subtries to be mapped onto different directions, and (2) two trie nodes on the same trie level to be mapped onto different stages.

4.3.3 Subtrie Inversion

In a trie, there are few nodes at the top levels while there are a lot of nodes at the leaf level. Hence, we can invert some subtries so that their leaf nodes are mapped onto the first several stages. We propose several heuristics to select the subtries to be inverted:

1. *Largest leaf*: The subtrie with the most number of leaves is preferred. This is straightforward since we need enough nodes to be mapped onto the first several stages:

2. *Least height*: The subtrie of shortest height is preferred. Due to *Constraint 1*, a subtrie with a larger height has less flexibility to be mapped onto pipeline stages.

3. *Largest leaf per height*: This is a combination of the previous two heuristics, by dividing the number of leaves of a subtrie by its height.

4. *Least average depth per leaf*: Average depth per leaf is the ratio of the sum of the depth of all the leaves to the number of leaves. This heuristic prefers a more balanced subtrie. A balanced subtrie has many nodes not only at the leaf level but also at the lower levels, which can help balance not only the first stage but also the first several stages.

Algorithm 2 finds the subtries to be inverted, where IFR denotes the *inversion factor*. A larger inversion factor results in more subtries to be inverted. When the inversion factor is 0, no subtrie is inverted. When the inversion factor is close to the pipeline depth, all subtries are inverted. The complexity of this algorithm is $O(K)$ where K denotes the total number of subtries.

Algorithm 2 Selecting the subtrie to be inverted

Input: K subtries.
Output: V subtries to be inverted.
1: N = total # of trie nodes of all subtries, H = # of pipeline stages, $V = 0$.
2: **while** $V < K < IFR \times \lceil N/H \rceil$ **do**
3: Based on the chosen heuristic, select one subtrie from those not inverted.
4: $V = V + 1$, $K = K - 1 +$ # of leaves of the selected subtrie.
5: **end while**

4.3.4 Mapping Algorithm

Now we have two sets of subtries. Those subtries which are mapped from roots are called the *forward subtries*, while the others are called the *reverse subtries*. We use a bidirectional fine-grained mapping algorithm (Algorithm 3). The nodes are popped out of the *ReadyList* in the decreasing order of their priority. The priority of a trie node is defined as its height if the node is in a forward subtrie, and its depth if in a reverse subtrie. The node whose priority is equal to the number of the remaining stages is regarded as a *critical* node. For the forward subtries, a node is pushed into the $NextReadyList$ immediately after its parent is popped. For the reverse subtries, a node will not be pushed into the $NextReadyList$ until all its children are popped. The complexity of this mapping algorithm is $O(HN)$ where H denotes the pipeline depth and N the total number of nodes.

4.3.5 Implementation Issues

We propose a method to enable two nodes on the same level of a subtrie to be mapped to different stages. Each node stored in the local memory of a pipeline stage has two fields. One is the distance to the pipeline stage where its child node is stored. The other is the memory address of the child node in that stage. Before a packet moves onto the next stage, its distance value is checked. If the value is zero, the memory address of its child node is used to index the memory in the next stage to retrieve its child node content. Otherwise, the packet will do nothing in that stage but decrement its distance value by one.

The effectiveness of the bidirectional mapping scheme is evaluated in Section 6.

Algorithm 3 Bidirectional fine-grained mapping

Input: K forward subtries.
Input: V reverse subtries.
Output: H stages with mapped nodes.
1: Create and initialize two lists: $ReadyList = \phi$ and $NextReadyList = \phi$.
2: R_n = # of remaining nodes, R_h = # of remaining stages $= H$.
3: Push the roots of the forward subtries and the leaves of the reverse subtries into *ReadyList*.
4: **for** $i = 1$ to H **do**
5: $M_i = 0$, $Critical = FALSE$.
6: Sort the nodes in *ReadyList* in the decreasing order of the node priority.
7: **while** $Critical = TRUE$ or ($M_i < \lceil R_n/R_h \rceil$ and $Readylist \neq \phi$) **do**
8: Pop node from *ReadyList* and map into Stage i.
9: **if** The node is in forward subtries **then**
10: The popped node's children are pushed into $NextReadyList$.
11: **else if** All children of the popped node's parent have been mapped **then**
12: The popped node's parent is pushed into $NextReadyList$.
13: **end if**
14: $Critical = FALSE$.
15: **if** There exists a node $N_c \in ReadyList$ and the priority of $N_c >= R_h - 1$ **then**
16: $Critical = TRUE$.
17: **end if**
18: **end while**
19: $R_n = R_n - M_i$, $R_h = R_h - 1$.
20: Merge the $NextReadyList$ to the $ReadyList$.
21: **end for**

5. TRAFFIC BALANCING AND INTRA-FLOW PACKET ORDER PRESERVING

This section studies traffic balancing among multiple pipelines while preserving the intra-flow packet order.

5.1 Traffic Balancing by Caching

Since the routing table is partitioned, it is possible for multiple IP packets mapped onto a same subtrie and simultaneously dispatched to the same entrance of a pipeline. To handle such an access conflict, we need a multi-port queue for each pipeline entrance. However, due to the TCP mechanism and application characteristics, Internet traffic contains a great amount of locality, resulting in bursts [12]. The queues will easily overflow in the case of bursty traffic.

On the other hand, caching has been proven to be an efficient mechanism to exploit Internet traffic locality to balance the load among parallel engines [1]. In our architecture shown in Figure 2, some small caches are added to cache the most recently searched IP addresses. When a new packet arrives, if it has a cache hit, it will skip lookup and be directed to the pipeline entrance whose queue has the fewest packets. If the packet has a cache miss, it must go to the appropriate pipeline entrance to traverse the corresponding subtrie. The cache can be organized in any associativity. We use full associativity as the default. The cache update is triggered, either when there is a route update that is related to some cached entry, or after a packet that previously had a cache miss retrieves its search result from the pipeline. Any replacement algorithm can be used to update the cache. The Least Recently Used (LRU) algorithm is used as the default.

5.2 Preserving Intra-Flow Packet Order

The packet within a flow may go out of order, due to caching and queuing. For instance, consider packets A, B, C belonging to the same flow and assume they arrive in that order. A is the first packet and it has a cache miss. Then it goes to the pipeline and completes lookup. Before the cache is updated, B arrives and has a cache miss. B is directed to the pipeline entrance whose queue may be almost full. Right after B enters the queue, the caches are updated and C arrives with a cache hit. C is dispatched to a queue with the fewest packets. At this time, if B has already entered the pipeline, C cannot catch up with B, since C must go through the pipeline, though it won't perform any lookup operation.

However, if B is still waiting in a long queue while C enters an almost empty queue, C will be output before B.

The schedulers can easily detect such intra-flow out-of-order packets while dispatching packets. The schedulers track the status of each queue, i.e. the number of packets waiting in the queue. When a packet has a cache hit, the scheduler dispatches it to the queue with the fewest packets. We call this queue the *lightest loaded queue*. Assume there are L_{min} packets in the lightest loaded queue. The scheduler also checks the status of the queue which corresponds to the subtrie onto which the packet is mapped. The mapping relationship is stored in the DIT. Assume there are L_c packets in this queue. The scheduler attaches a delay value of $L_c - L_{min}$ to the packet. As shown in Figure 2, a packet goes to an output delay queue when it exits the pipeline. Each output delay queue is built as a single-entrance multi-exit pipeline. If the delay value of the packet is 0, the packet is output immediately. Otherwise, the packet goes through the output delay queue and decrements its delay value by one at each stage, until its delay value becomes zero. Note that it is possible to output multiple packets of a flow at the same time, in case these packets are input into the architecture simultaneously. To preserve the order among these packets, they are tagged with a value of $1 \sim W$ at the Front End and output in the tag order.

We call the queues for each pipeline entrance the *input queues* and the output delay queues the *output queues*. We can see from the above discussion that the size of the output queue should be equal to that of the input queue. We will also see later that the queue size needed is quite small.

6. PERFORMANCE EVALUATION

This section evaluates the performance of the architecture. First, we examine the memory balancing across all pipeline stages by using the proposed schemes. Then, we measure the throughput using real-life traffic traces. All experiments are based on simulation.

6.1 Memory Balancing across Pipeline Stages

We conducted experiments on the four routing tables in Table 2 to evaluate the effectiveness of the bidirectional fine-grained mapping scheme. In these experiments, the initial stride used for partitioning the trie was $I = 12$. Other parameters were set as $P = 4, H = 25$.

When the inversion factor is 0, no subtrie is inverted and hence the first several stages cannot be balanced. When the inversion factor is 25, i.e. the pipeline depth, all subtries are inverted and the last several stages cannot be balanced. Hence we need to select an appropriate value for the inversion factor. We conducted the experiments changing the inversion factor from 0 to 25. We found that when the inversion factor was in the range of $4 \sim 8$, the node distribution across all pipeline stages was perfectly balanced, regardless of which inversion heuristic was used. Figure 5 shows the results of mapping the routing tables onto four 25-stage pipelines, with the inversion factor of 4 and the *Least average depth per leaf* inversion heuristic.

We also examined the impact of the proposed four inversion heuristics, when the inversion factor was set to 1. We found that the *least average depth per leaf* heuristic had the best performance, showing that when we have a choice, a balanced subtrie should be inverted. Due to space limitation, we do not present detailed results here.

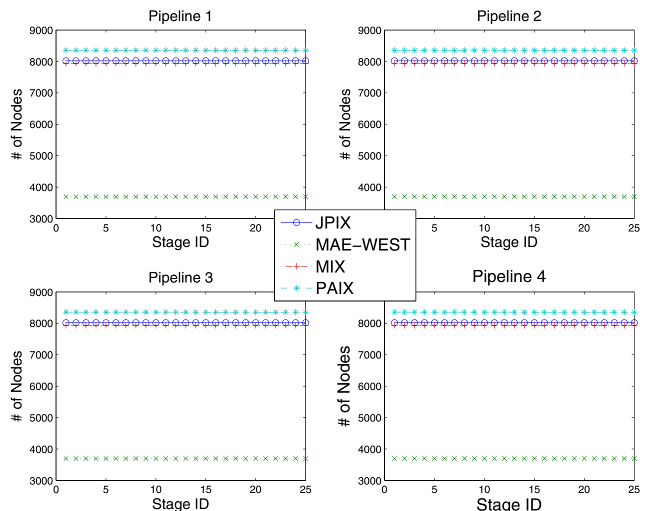


Figure 5: Node distribution over four 25-stage pipelines. (Inversion factor = 4, *Least average depth per leaf* heuristic)

6.2 Throughput Scaling

We used real-life Internet traffic traces to evaluate the throughput performance of the architecture. Two anonymized real-life traces were collected from [15]. Their information is listed in Table 3. Due to the unavailability of public IP traces associated with their corresponding routing tables, we generated the routing tables by extracting the unique destination IP addresses from the traces.

In these experiments, the default setting of the architecture parameters was $W = 8, P = 4, H = 25, Q = 16, C = 160$. The performance metric is the throughput in terms of the number of packets processed per clock cycle (PPC). Note that in a W-width architecture, the throughput is $\leq W$.

We increased the number of pipelines while keeping the input width $W = 2P$, and observed the throughput scalability. As shown in Figure 6, with caching, the throughput scaled well with the number of pipelines. When there were 4 pipelines, the throughput could be as high as 7.5 PPC. On the other hand, the overhead was low: only 1% of routing entries were cached, and the queue size was quite small as well. We did more experiments by increasing the queue size but did not obtain much improvement. Hence, small queues are sufficient in our architecture. In all the above experiments, we also found that the packets within any flow were output in the same input order.

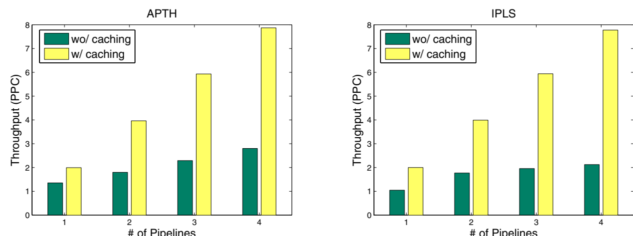


Figure 6: Throughput vs. # of Pipelines. ($H = 25, Q = 16, C = 160$.)

Table 3: IP header traces

Trace	Date	# of packets	# of unique IP entries
APTH: AMP-1110523221-1	20050311	769100	17628
IPLS: I2A-1091235138-1	20040731	1821364	15791

6.3 Overall Performance

Based on the previous experiments, we estimate the overall performance of a 8-width 4-pipeline 25-stage architecture. As Figure 5 shows, for the backbone routing table MIX with 236991 prefixes, each stage has no more than 7939 nodes. A 13-bit address is enough to index a node in the local memory of a stage. Since the pipeline depth is 25, we need an extra 5 bits to specify the distance. Thus, each node stored in the local memory needs 18 bits. The total memory needed is $18 \times 2^{13} \times 25 \times 4 \approx 14.75$ Mb ≈ 1.8 MB, where each stage needs 18 KB of memory. We use CACTI 4.2[4] to estimate the memory access time and the power consumption. An 18-KB dual-port SRAM using 45 nm technology needs 0.4 ns to access, and dissipates 0.008 W of power. The maximum clock rate of the above architecture in ASIC implementation can be 2.5 GHz. Considering the throughput of 7.5 PPC as shown in Section 6.2, the overall throughput can be as high as $7.5 \times 2.5 = 18.75$ G packets per second, i.e 6.0 Tbps for the minimum packet size of 40 bytes. Such a throughput is 35 times that of the state-of-the-art TCAM-based IP lookup engines [24]. The overall power consumption to complete one IP lookup is $0.008 \times 25 = 0.2$ W, a 10-fold reduction of that of the "cool" TCAM solution [23].

7. CONCLUSION

This paper proposed an SRAM-based multi-pipeline architecture for multi-terabit trie-based IP lookup. The architecture consists of multiple bidirectional linear pipelines, each of which stores part of a routing table. The architecture provides more flexibility for mapping a routing trie to the pipelines, so that the memory distribution over different pipelines as well as across different stages in each pipeline are both balanced. Furthermore, IP caching is effectively integrated to scale the throughput improvement. Using 1.8 MB of memory to store a core routing table with nearly 237K prefixes, the proposed 4-pipeline architecture can achieve a high throughput of up to 6 Tbps i.e. 37.5× the OC-3072 rate. Our future work includes prototyping the proposed architecture on FPGAs and evaluating its performance under real-life scenarios.

8. REFERENCES

[1] M. J. Akhbarizadeh, M. Nourani, R. Panigrahy, and S. Sharma. A TCAM-based parallel architecture for high-speed packet forwarding. *IEEE Trans. Comput.*, 56(1):58–72, 2007.

[2] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. In *Proc. ISCA*, pages 123–133, 2005.

[3] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. In *Proc. INFOCOM*, pages 64–74, 2003.

[4] CACTI 4.2. http://quid.hpl.hp.com:9081/cacti/.

[5] Cypress Sync SRAMs. http://www.cypress.com.

[6] W. Jiang and V. K. Prasanna. A memory-balanced linear pipeline architecture for trie-based IP lookup. In *Proc. Hot Interconnects (HotI '07)*, pages 83–90, 2007.

[7] W. Jiang, Q. Wang, and V. K. Prasanna. Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup. In *Proc. INFOCOM*, 2008.

[8] Juniper Networks T1600 Core Router. http://www.juniper.net.

[9] K. S. Kim and S. Sahni. Efficient construction of pipelined multibit-trie router-tables. *IEEE Trans. Comput.*, 56(1):32–43, 2007.

[10] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[11] S. Kumar, M. Becchi, P. Crowley, and J. Turner. CAMP: fast and efficient IP lookup architecture. In *Proc. ANCS*, pages 51–60, 2006.

[12] S. Kumar, J. Maschmeyer, and P. Crowley. Exploiting locality to ameliorate packet queue contention and serialization. In *Proc. Computing Frontiers (CF '06)*, pages 279–290, 2006.

[13] W. Lu and S. Sahni. Packet forwarding using pipelined multibit tries. In *Proc. ISCC*, 2006.

[14] S. A. McKee. Reflections on the memory wall. In *Proc. Computing Frontiers (CF '04)*, page 162, 2004.

[15] NLANR network traffic packet header traces. http://pma.nlanr.net/traces/.

[16] Renesas CAM ASSP Series. http://www.renesas.com.

[17] RIS Raw Data. http://data.ris.ripe.net.

[18] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network*, 15(2):8–23, 2001.

[19] SAMSUNG High Speed SRAMs. http://www.samsung.com.

[20] L. Shi, Y. Zhang, J. Yu, B. Xu, B. Liu, and J. Li. On the extreme parallelism inside next-generation network processors. In *Proc. INFOCOM*, pages 1379–1387, 2007.

[21] A. Singhal and R. Jain. Terabit switching: a survey of techniques and current products. *Comput. Communications*, 25:547–556, 2002.

[22] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Trans. Comput. Syst.*, 17:1–40, 1999.

[23] F. Zane, G. J. Narlikar, and A. Basu. CoolCAMs: Power-efficient TCAMs for forwarding engines. In *Proc. INFOCOM*, pages 42–52, 2003.

[24] K. Zheng, C. Hu, H. Lu, and B. Liu. A TCAM-based distributed parallel IP lookup scheme and performance analysis. *IEEE/ACM Trans. Netw.*, 14(4):863–875, 2006.