# Parallel IP Lookup using Multiple SRAM-based Pipelines

Weirong Jiang and Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, USA
{weirongj, prasanna}@usc.edu

## Abstract

*Pipelined SRAM-based algorithmic solutions have become competitive alternatives to TCAMs (ternary content addressable memories) for high throughput IP lookup. Multiple pipelines can be utilized in parallel to improve the throughput further. However, several challenges must be addressed to make such solutions feasible. First, the memory distribution over different pipelines as well as across different stages of each pipeline must be balanced. Second, the traffic among these pipelines should be balanced. Third, the intra-flow packet order should be preserved. In this paper, we propose a parallel SRAM-based multi-pipeline architecture for IP lookup. A two-level mapping scheme is developed to balance the memory requirement among the pipelines as well as across the stages in a pipeline. To balance the traffic, we propose a flow pre-caching scheme to exploit the inherent caching in the architecture. Our technique uses neither a large reorder buffer nor complex reorder logic. Instead, a payload exchange scheme exploiting the pipeline delay is used to maintain the intra-flow packet order. Extensive simulation using real-life traffic traces shows that the proposed architecture with 8 pipelines can achieve a throughput of up to 10 billion packets per second (GPPS) while preserving intra-flow packet order.*

## 1 Introduction

As the Internet continues to grow rapidly, IP lookup with longest prefix matching becomes a major performance bottleneck for backbone routers. Advances in optical networking technology are pushing link rates in high speed IP routers beyond OC-768 (40 Gbps). Such high rates demand that IP lookup in routers be performed in hardware. For instance, 40 Gbps links require a throughput of 8 ns per lookup, i.e. 125 million packets per second (MPPS), for a minimum size (40 bytes) packet. Such throughput is impossible using existing software-based solutions [19].

There are two main hardware-based solutions for high speed IP lookup: 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 speed of TCAMs. They are expensive and offer little flexibility to adapt to new addressing and routing protocols [2]. As shown in Table 1, SRAM outperforms TCAM 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 [19], a tree-like data structure representing a collection of prefixes, is widely used in DRAM/SRAM-based solutions. It needs multiple memory accesses to search a trie to find the longest matched prefix for an IP address.

Pipelining has been explored to improve significantly the throughput of trie-based IP lookup. 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 [4]. 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 for the following aspects. First, it needs more time to access the larger local memory. This leads to 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 stage, we need to allocate memory with the maximum size for each stage. Such an over-provisioning results in memory wastage [3]. To achieve a balanced memory dis-

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

**Table 1. Comparison of TCAM and SRAM technologies**

| | TCAM (18 Mbits Chip) | SRAM (18 Mbits Chip) |
|---|---|---|
| Advertised maximum clock rate (MHz) | 266 [17] | 400 [6, 20] |
| Cell size (# of transistors per bit) | 16 | 6 |
| Power consumption (Watts) | $12 \sim 15$ [24] | $\approx 0.1$ [5] |

tribution across stages, several novel pipeline architectures have been proposed [3,13]. However, their non-linear structures result in throughput degradation, and most of them must disrupt ongoing operations during a route update. Our previous work [9] proposed a linear pipeline architecture with a fine-grained node-to-stage mapping scheme to distribute nodes of a leaf-pushed uni-bit trie evenly to different pipeline stages. It can achieve a high throughput of one lookup per clock cycle. Also, it can support *write bubble*s, as proposed in [4], for incremental updates without disrupting router operations.

However, projected improvements in memory access speed are rather limited. Thus it becomes necessary to employ multiple pipelines operating in parallel to speed up 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. 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 works on parallel TCAM-based IP lookup engines use either a learning algorithm to predict the future behavior of incoming traffic based on its current distribution [25], or IP/prefix caching to utilize the locality of Internet traffic [1, 14]. Nevertheless, neither of them can work efficiently in deep pipeline architectures, since the long pipeline delay results in slow feedback on either distribution prediction or cache updating. In addition, most of existing schemes make the packets within a flow out of order, and thus need to add large reorder buffers and complicated reorder logic to preserve the intra-flow packet order.

To address the above problems, this paper proposes a partition-based parallel IP lookup engine using multiple SRAM-based linear pipelines. Our work makes the following contributions.

- To the best of our knowledge, this work is among the first discussions of SRAM-based multi-pipeline solutions for parallel IP lookup.
- A novel two-level mapping scheme is proposed to balance the memory distribution over multiple linear pipelines as well as across all pipeline stages.
- A *flow pre-caching* scheme is developed to balance the load among pipelines. In contrast to the traditional caching schemes which suffer from pipeline delay, our scheme benefits from deep pipelining since it utilizes the inherent caching in the architecture.
- An approach called *payload exchange*, which exploits the pipeline delay, is used to maintain the intra-flow packet order. Neither a large reorder buffer nor complex reorder logic is needed.
- Our results demonstrate that SRAM-based pipelined algorithmic solutions are a promising alternative to TCAMs for future high-end routers. The proposed 8-pipeline architecture can store a full backbone routing table with over 200K unique prefixes using 3.6 MB of memory. It can achieve a high throughput of up to 10 billion packets per second, i.e. 3.2 Tbps for minimum size (40 bytes) packets.

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

## 2 Background

### 2.1 Trie-based IP Lookup

The basic mission of IP lookup is longest prefix matching (LPM). The most common data structure in algorithmic solutions for doing LPM is some form of trie [19]. 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 decisions at a time. The prefix set in Figure 1 (a) corresponds to the uni-bit trie in Figure 1 (b). For

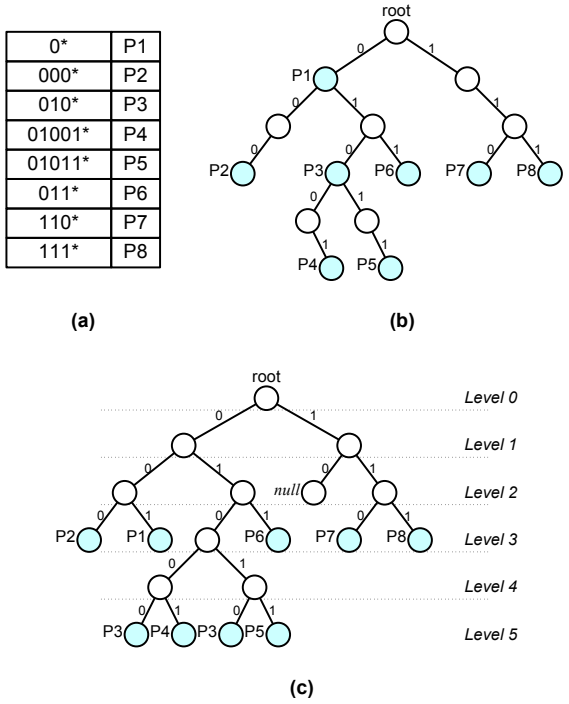


**Figure 1. (a) Prefix set; (b) Uni-bit trie; (c) Leaf-pushed uni-bit trie.**

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* [21], 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).

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 the IP address. 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*. Some optimization schemes [7, 11] have been proposed to build a memory-efficient multi-bit trie. 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

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

Basu et al. [4] and Kim et al. [11] both reduce the memory imbalance 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 [11], Lu et al. [15] propose a tree-packing heuristic to balance the memory further, but it does not solve the fundamental problem of how to retrieve a node's descendants that 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 [4].

Baboescu et al. [3] propose a Ring pipeline architecture for trie-based IP lookup. The memory stages are configured in a circular, multi-point access pipeline so that lookups can be initiated at any stage. The trie is split into several small subtries of equal size. These subtries are then mapped to different stages to create a balanced pipeline. Some subtries must wrap around if their roots are mapped to the last several stages. Though all IP packets enter the pipeline from the first stage, their lookup processes may be activated at different stages. Hence all the IP lookup packets must traverse the pipeline twice to complete the trie traversal. The throughput is thus 0.5 lookups per clock cycle. Kumar et al. [13] extend the circular pipeline with a new architecture called the Circular, Adaptive and Monotonic Pipeline (CAMP). CAMP has multiple entry and exit points, resulting in out-of-order output and delay variation. Several *request queue*s are employed to manage the access conflicts between the new request and the one from the preceding stage. CAMP can achieve a worst-case throughput of 0.8 lookups per clock cycle, while maintaining nearly balanced memory across pipeline stages. Due to the non-linear structure, neither the Ring pipeline nor CAMP supports well the *write bubble* proposed in [4] for the incremental route update. Our previous work [9] proposes an optimized linear pipeline architecture, named OLP, to achieve a throughput of one output per clock cycle while allowing the insertion of *write bubble*s. By supporting *nop*s (no-operations) in the pipeline, it offers more freedom in mapping trie nodes to pipeline stages. This paper extends the idea of OLP to map a trie to multiple pipelines, while keeping the memory requirement balanced across all stages.

## 2.3 Partition-based Parallel IP Lookup Engines

Most published parallel IP lookup engines are TCAM-based. They partition the full routing table into several

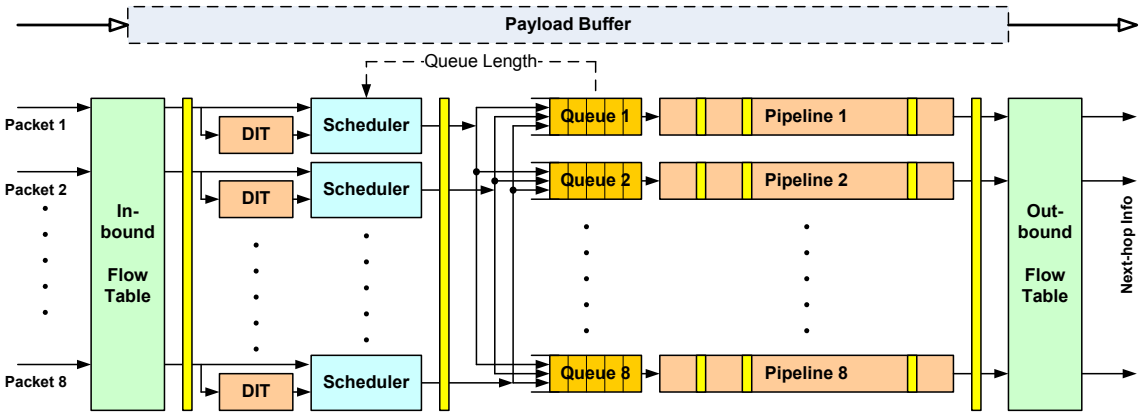


**Figure 2. Block diagram of the architecture ($P = 8$).**

blocks, and make the search process parallel on different blocks. Both power efficiency and throughput improvement can be obtained by such partitioning and parallelization.

There are two methods to partition the routing table [24]: bit-selection and trie-based approaches. In the former, selected bits are used to index different TCAM blocks directly. However, prefix distribution imbalance among the TCAM blocks may be quite high, resulting in low worst-case performance [14]. The latter scheme splits the trie by carving subtries out of the full trie. This can have a much better worst-case bound. Subtrie-to-block mapping is implemented using an index logic consisting of a TCAM and an SRAM. 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 have been proposed, including learning-based block rearrangement [25] and IP/prefix caching [1, 14]. 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. The latter exploits the Internet traffic locality effectively to speed up the throughput. However, contrary to TCAMs, pipeline solutions need several clock cycles to retrieve lookup results. The cache miss penalty may be quite high due to the large processing delay [22]. It is even worsened by deeper pipelining with larger delay.

In load-balanced parallel packet processing engines, packets within a flow may go out of order due to caching and queuing. It affects adversely some network applications [8, 23]. Thus reorder buffers and logic are usually needed, which are expensive and complicated. This paper aims to eliminate them, by exploiting the processing delay in the pipelines.

## 3 Architecture Overview

The proposed SRAM-based parallel architecture with multiple memory-balanced linear pipelines is shown in Figure 2. It does not employ any TCAM. It can be fundamentally divided into two parts based on the functions: lookup engines and load balancer.

### 3.1 Lookup Engines

The architecture consists of $P$ pipelines, each of which stores part of the full routing table. Figure 2 shows an architecture with $P = 8$. All pipelines have the same number of stages. The routing table is constructed as a leaf-pushed uni-bit trie. The trie is partitioned into disjoint subtries using initial bits of the prefixes. We propose a polynomial-time approximation algorithm to map the subtries to pipelines. It balances the memory requirement over different pipelines. Within each pipeline, a fine-grained node-to-stage mapping is employed to balance the trie node distribution across the stages. The details of memory balancing are discussed in Section 4.

To store the mapping function between subtries and pipelines, several small memories called Destination Index Tables (DITs) are used. Initial bits of the IP address of an incoming packet are used to index DITs, to retrieve the pipeline ID to which the packet will be routed. A packet is directed to the pipeline that stores its corresponding subtrie. By searching the DIT, the packet also retrieves the address of the subtrie's root in the first stage of the pipeline. $P$ DITs can be used in parallel to process $P$ packets simultaneously. Each pipeline employs a multi-port queue to handle the access conflicts when multiple incoming packets are directed

to the same pipeline.

### 3.2 Load Balancer

Caching is an efficient way to exploit Internet traffic locality for parallel IP lookup. To relieve the cache miss penalty due to large pipeline delay, our architecture extends the idea of *flow caching* from Layer-4 switching, where only the first packet of a flow needs lookup, and the rest of the packets of the flow are cut-through routed through flow cache entry lookups [22]. In this paper we define a sequence of packets with the same destination IP address as a *flow* [1]. We propose a scheme called *flow pre-caching*, which allows the destination IP address of a flow to be cached before its next-hop information is retrieved. When a new packet arrives, it compares its destination IP address with the cached IP addresses. If the arriving packet matches any of the cached flows, it will be assigned the ID of that flow no matter whether the next-hop information of that flow is available or not. In other words, the new packet pre-fetches its lookup result even if its flow has not retrieved the next-hop information. The Scheduler directs this packet to the pipeline with minimum load (whose queue has the fewest packets). Then, this packet goes through the pipeline without any operation. Otherwise (i.e. the new packet does not match any of the cached flows), it is treated as the first packet of a new flow. The Scheduler directs it to the pipeline whose ID is obtained through indexing DITs. Then this packet goes through the pipeline to perform the lookup. When a packet exits pipelines, it uses its flow ID to index the Outbound Flow Table to find its next-hop information. If there is no valid information in the Outbound Flow Table for it, the next-hop information retrieved from pipelines will be used to update that flow entry.

The intra-flow packets may go out of order due to caching and queuing. However, in our architecture, all packets are required to go through the pipelines from the first stages, no matter whether they have cache hit or miss. Thus the queued packets cannot catch up with their preceding packets that are already in the pipelines. Thus the Scheduler can detect the intra-flow out-of-order packets when sending packets to queues. If the intra-flow out-of-order packet is detected, a task to exchange the payload between out-of-order packets is initiated. As it takes multiple clock cycles for a packet to complete looking up the pipelines, the payload exchange has enough time to be completed before the packets exit the pipelines. Thus, the intra-flow packet order can be preserved.

The details of both flow pre-caching and payload exchange schemes are discussed in Section 5.

[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 [22].

## 4 Memory Balancing

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

1. Partitioning the entire routing table in a simple but efficient way without TCAMs;
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.

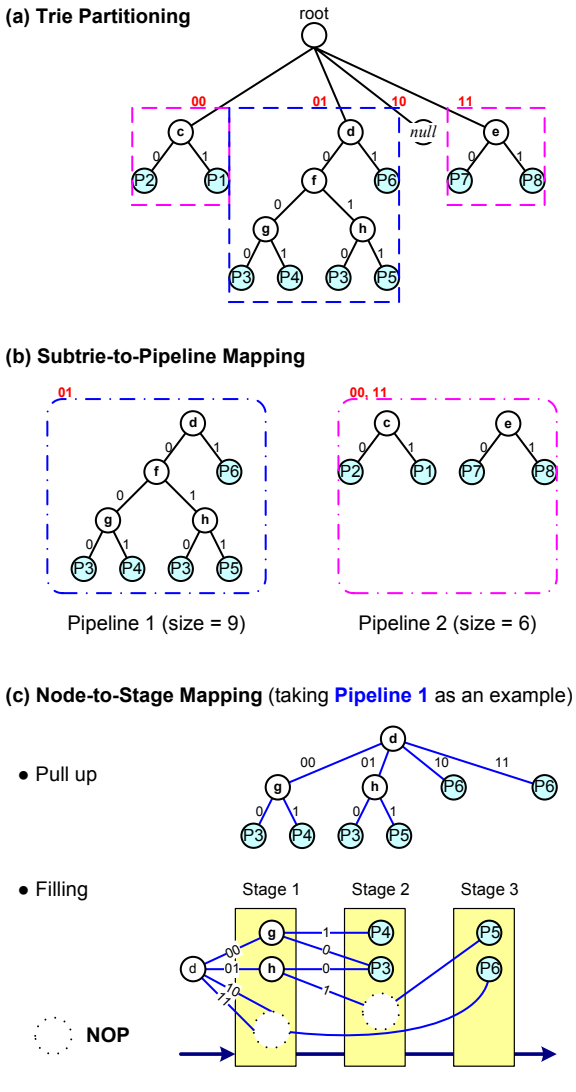


**Figure 3. Memory balancing ($I = 2, P = 2, H = 3$).**

**Table 2. Representative Routing Tables**

| Routing table | Location | Date | # of prefixes | # of prefixes with length $< 16$ |
|---|---|---|---|---|
| RIPE NCC(rrc00) | Amsterdam, Netherlands | 20071130 | 243474 | 1949 (0.80%) |
| LINX (rrc01) | London, UK | 20071130 | 240797 | 1945 (0.81%) |
| SFINX (rrc02) | Paris, France | 20071130 | 238089 | 1941 (0.82%) |
| AMS-IX (rrc03) | Amsterdam, Netherlands | 20071130 | 246530 | 1950 (0.79%) |
| CIXP (rrc04) | Geneva, Switzerland | 20071130 | 240180 | 1948 (0.81%) |
| VIX (rrc05) | Vienna, Austria | 20071130 | 241948 | 1968 (0.81%) |
| JPIX (rrc06) | Otemachi, Japan | 20071130 | 239332 | 1926 (0.80%) |
| NETNOD (rrc07) | Stockholm, Sweden | 20071130 | 248856 | 1943 (0.78%) |
| MAE-WEST (rrc08) | San Jose, USA | 20040901 | 83556 | 495 (0.59%) |
| TIX (rrc09) | Zurich, Switzerland | 20040201 | 132786 | 991 (0.75%) |
| MIX (rrc10) | Milan, Italy | 20071130 | 236991 | 1939 (0.82%) |
| NYIIX (rrc11) | New York, USA | 20071130 | 238836 | 1952 (0.82%) |
| DE-CIX (rrc12) | Frankfurt, Germany | 20071130 | 243731 | 1999 (0.82%) |
| MSK-IX (rrc13) | Moscow, Russia | 20071130 | 238461 | 1942 (0.81%) |
| PAIX (rrc14) | Palo Alto, USA | 20071130 | 243731 | 1949 (0.80%) |
| PTTMetro-SP (rrc15) | Sao Paulo, Brazil | 20071130 | 243242 | 1946 (0.80%) |

We use the following definitions.

**Def. 1.** *Two subtries are **disjoint** if they share no prefix.*

**Def. 2.** *The **size** of a trie is the number of nodes in it.*

**Def. 3.** *The **depth** of a pipeline is the number of stages in it.*

**Def. 4.** ***Height** of a trie node is the maximum directed distance from it to a leaf node.*

**Def. 5.** *The **descent size** of a trie node is the number of its descendant nodes. For example, in Figure 3, the descent size of node **d** is 8.*

## 4.1 Partitioning Routing Table

To partition the trie, we adopt a scheme called prefix expansion [21], illustrated in Figure 3 (a). 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), prefix P3 whose length is 3 will be copied to two subtries. One subtrie with the initial bits of "0100" has prefixes P3 and P4, and the other with "0101" has prefixes P3 and P5.

We study the prefix length distribution based on sixteen BGP backbone routing tables, rrc00 $\sim$ rrc15, collected from [18]. As shown in Table 2, few prefixes are shorter than 16. We also conduct experiments by varying $I$. When $I \leq 8$, it does not result in any prefix duplication and guarantees all the resulting subtries are disjoint. In following sections, we pick $I = 8$ as a default.

## 4.2 Subtrie-to-Pipeline Mapping

Our partitioning scheme may result in many subtries of various sizes. For example, using $I = 8$ to partition the tries corresponding to the above 16 routing tables, some large subtries have over 30K nodes while some small subtries have only 1 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. It can be formulated as (1).

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

with the constraint (2):

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

where $P$ denotes the number of pipelines; $S_i$ the set of subtries contained by the $i$th pipeline, $i = 1, 2, \cdots, P$; $K$ the number of subtries; $T_i$ the $i$th subtrie, $i = 1, 2, \cdots, K$; and $size(.)$ denotes the size of a subtrie or a set of subtries.

#### 4.2.2 Mapping Algorithm

The above optimization problem is *NP-complete*. This can be shown by a reduction from the partitioning problem [12]. To solve it, we use a polynomial-time approximation algorithm shown in Algorithm 1. According to [12], in the worst-case, the resulting largest pipeline may have 1.5 times the number of nodes as in the optimal mapping. Figure 3 (b) illustrates an example of mapping 3 subtries to 2 pipelines. The complexity of this algorithm is $O(KP)$.

**Algorithm 1** Subtrie-to-pipeline mapping

**Input:** $K$ subtries: $\{T_i | i = 1, 2, \cdots, K\}$; and $P$ 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 executed the algorithm on the 16 routing tables given in Table 2. In these experiments, we set $P = 8$. We obtained the resulting size of each pipeline as shown in Figure 4.

According to Figure 4, for all the 16 routing tables, our algorithm resulted in balanced memory distribution among 8 pipelines.

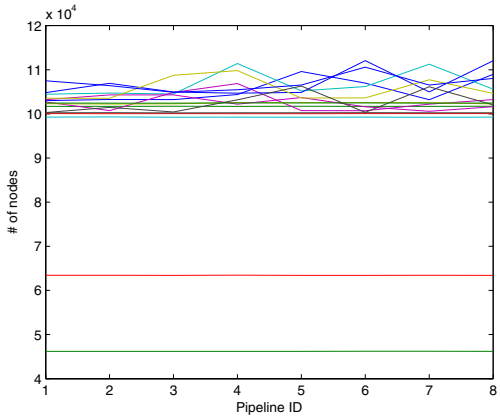


**Figure 4. Node distribution over 8 pipelines.**

### 4.3 Node-to-Stage Mapping

#### 4.3.1 Problem Formulation

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 stages balanced. Also, each pipeline should be linear. The problem is formulated as (3):

$$\min \max_{i=1,2,\cdots,H} M_i \tag{3}$$

with the constraints (4), (5) and (6):

$$\sum_{i=1}^{H} M_i = \sum_{j=1}^{L} size(T_j) \tag{4}$$

*Constraint* (5): All the subtries' roots are mapped to the first stage;

*Constraint* (6): If node $A$ is an ancestor of node $B$ in a subtrie, then $A$ must be mapped to a stage preceding the stage to which $B$ is mapped;

where $H$ denotes the pipeline depth; $M_i$ the number of nodes mapped to the $i$th stage, $i = 1, 2, \cdots, H$; $L$ the number of subtries assigned to the pipeline; $T_i$ the $i$th subtrie, $i = 1, 2, \cdots, L$; and $size(.)$ denotes the size of a subtrie.

#### 4.3.2 Mapping Algorithm

We use a simple heuristic to perform the node-to-stage mapping. As Figure 3 (c) shows, by supporting $nop$s, we allow the nodes on the same level of a subtrie to be mapped onto different pipeline stages. This provides more flexibility to map the trie nodes and helps achieve a balanced node distribution across the stages in a pipeline.

Before mapping, two problems must be addressed. First, since the top levels of a subtrie have fewer nodes, it is difficult to balance the first several stages with other stages. Second, when the number of trie levels exceeds the number of pipeline stages, it is impossible to map all nodes onto the pipeline. In either case, we again use the scheme of prefix expansion [21] to pull up nodes. Figure 3 (c) illustrates the pull-up process using a stride of 2.

Now we map trie nodes to the pipeline stages while satisfying *Constraint* (6). We manage two lists, namely $ReadyList$ and $NextReadyList$. The former stores the nodes that are available for filling the current stage, while the latter stores the nodes for filling the next stage. Since Stage 1 is dedicated for (possibly pulled-up) subtries' roots, we start with mapping the nodes that are children of the roots onto Stage 2. When filling a stage, the nodes in $ReadyList$ are popped out and mapped onto the stage, in the decreasing order of their heights. If a node is assigned to a stage, its children are pushed into the $NextReadyList$.

Meanwhile, we check whether there are any $Critical$ nodes that must be mapped onto the current stage. A $Critical$ node is defined as the node whose height is larger than the number of remaining stages. If such a node is not mapped onto the current stage, none of its descendants can be mapped later. When a stage is full or $ReadyList$ becomes empty, we move on to the next stage. At that time, the $NextReadyList$ is merged into $ReadyList$. By these means, *Constraint* (6) can be met. The complete algorithm is shown in Algorithm 2, where $R_n$ denotes the number of remaining nodes to be mapped onto stages, and $R_h$ the number of remaining stages for nodes to be mapped onto. The complexity of this mapping algorithm is $O(HN)$ where $H$ denotes the pipeline depth and $N$ the total number of trie nodes.

**Algorithm 2** Node-to-stage mapping

**Input:** $L$ subtries: $\{T_i | i = 1, 2, \cdots, L\}$; and $H$ empty stages.
**Output:** $H$ stages with mapped nodes.

```
1: Initialization: ReadyList = φ, NextReadyList = φ,
   R_n = Σ_{i=1,2,...,L} size(T_i), R_h = H.
2: Map the roots of the subtries into Stage 1. Push the
   children of the mapped nodes into ReadyList.
3: R_n = R_n − M_1, R_h = R_h − 1.
4: for i = 2 to H do
5:    Sort the nodes in ReadyList in the decreasing order
      of their heights. If two nodes have the same height,
      the node with the larger descent size is sorted prior
      to the other.
6:    while 1 do
7:       if M_i < R_n/R_h AND Readylist ≠ φ then
8:          Pop node from ReadyList.
9:       else if ∃CriticalNode then
10:         Pop the critical node from ReadyList.
11:      else
12:         Break.
13:      end if
14:      Map the popped node onto Stage i.
15:      Push its children into NextReadyList.
16:      M_i = M_i + 1.
17:   end while
18:   R_n = R_n − M_i, R_h = R_h − 1.
19:   Merge the NextReadyList to the ReadyList.
20: end for
```

#### 4.3.3 Implementation Issues

To allow two nodes on the same subtrie level to be mapped to different stages, we must implement the *nop* (no-operation) in the pipeline. We propose a simple method to enable this. Each node stored in the local memory of a pipeline stage has two fields. One is the memory address of its child node in the pipeline stage where the child node is stored. The other is the distance to the pipeline stage where the child node is stored. For example, when we search the prefix $0110$ in Figure 3, the first two bits $01$ direct the packet to Pipeline 1. Then we search the following bits from Stage 1. We will get (1) node P6's memory address in Stage 3, and (2) the distance from Stage 1 to Stage 3. When a packet is passed through the pipeline, the distance value is decremented by 1 when it goes through a stage. When the distance value becomes 0, the child node's address is used to access the memory in that stage.

#### 4.3.4 Experimental Results

We mapped the 16 routing tables onto 8 pipelines each of which has 25 stages. The trie node distribution over the stages is shown in Figure 5. Except for the first few stages, all the stages had almost equal numbers of trie nodes.

#### 4.3.5 Resource Estimation

There are 8 copies of DITs. Since $I = 8$, each DIT has $2^8$ entries. Each entry has 3 bits to indicate the 8 pipelines. Thus the total memory needed for DITs is $8 \times 2^8 \times 3 = 6.144$ Kb = $0.768$ KB, which is quite small.

Consider the largest routing table rrc07 among the 16 routing tables shown in Table 2. According to Figure 5, each stage has fewer than 8K nodes. Thus 13 address bits are enough to index a node in the local memory of a stage. The pipeline depth is 25, and thus we need 5 bits to specify the distance. Each node stored in the local memory needs 18 bits. The total memory needed to storing 248856 prefixes from rrc07 in a 25-stage 8-pipeline architecture is $18 \times 2^{13} \times 25 \times 8 \approx 28$ Mb $= 3.6$ MB, where each stage needs 18 KB of memory. Using CACTI 4.2 [5], we estimate the memory access time. A 18 KB SRAM using 90 nm technology needs fewer than 0.75 ns to access. In other words, the maximum clock rate of the above architecture can be over 1.33 GHz.

## 5 Traffic Balancing

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

### 5.1 Caching with Deep Pipelines

It has been shown that caching is an efficient mechanism to exploit Internet traffic locality to balance the load among parallel engines [14]. When a new packet arrives, if it has a cache hit, it will skip lookup. Otherwise, it needs to complete the lookup process. However, in SRAM-based deep

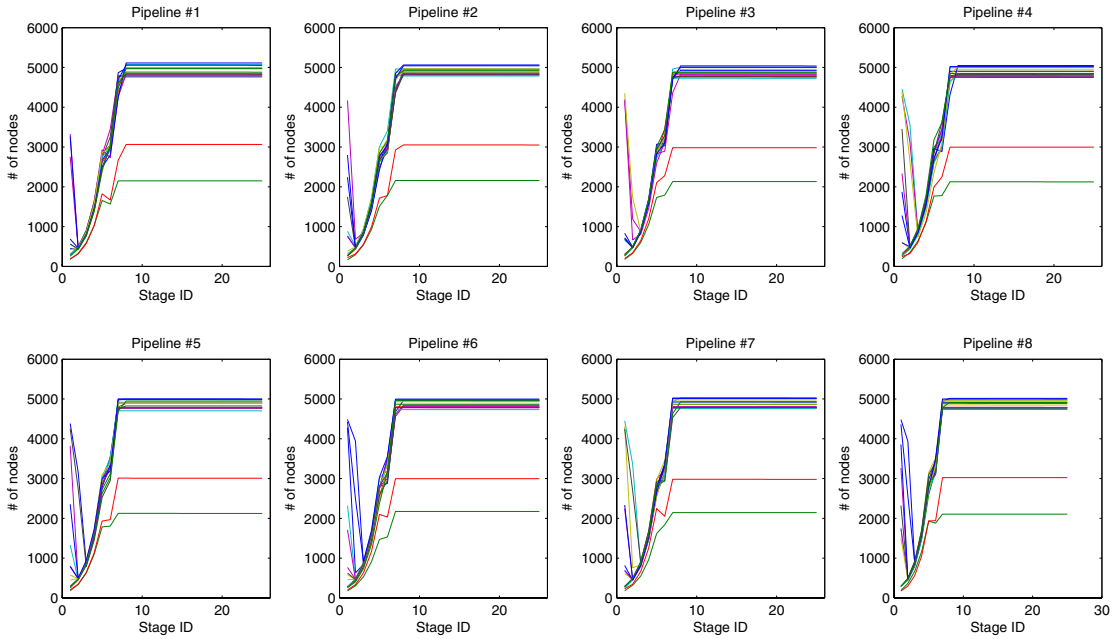


**Figure 5. Node distribution over stages.**

pipeline architectures, after a cache miss occurs, it takes a long time to retrieve the lookup result, and even longer time to update the caches. This results in two problems.

First, the caching may fail to capture the traffic locality due to the long pipeline delay. Consider the case when the pipeline depth is much larger than the burst length of a flow. All packets of the flow will have cache miss, since it takes a long time for the first packet of the flow to complete lookup.

Second, the intra-flow packets may go out of order. For example, 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 enters the pipeline to do lookup. Before A exits the pipeline, B arrives and has a cache miss. After A completes lookup and the cache is updated, B may be still in the pipeline. At this time, C arrives and has a cache hit. Thus C is outputted before B.

The above two problems can be further worsened when additional queues are employed, which makes the processing delay larger and unpredictable.

## 5.2 Flow Pre-Caching

The key idea of flow pre-caching is to allow the IP address of a flow to be cached before it has retrieved its next-hop information. As a result, the subsequent packets of the same flow can prefetch the lookup result as the first packet of the flow is being looked up in the pipeline.

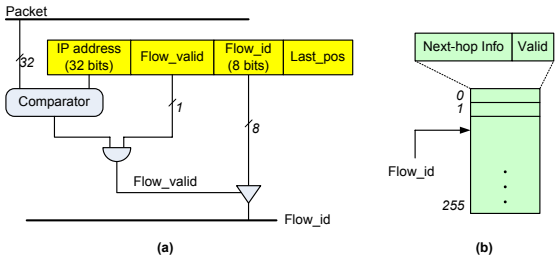


**Figure 6. (a) One entry in Inbound Flow Table; (b) Outbound Flow Table.**

We develop some logic called Inbound Flow Table and Outbound Flow Table shown in Figure 6. In a $P$-pipeline architecture, Inbound Flow Table is constructed as a $P$-port fully associative memory, while Outbound Flow Table is a $P$-port directly addressable memory. The information of each flow existing in the architecture is stored in a register in Inbound Flow Table, while Outbound Flow Table stores the next-hop information for each flow.

Each arriving packet compares with all flow entries in parallel. If the packet is the first packet of a flow, a new flow ID is assigned and a new flow entry is added into Inbound Flow Table. The flow ID is not valid until this packet enters the pipelines. If the packet is not the first packet of a flow, it can find its flow ID from Inbound Flow Ta-

ble directly. Thus, a packet can pre-fetch its lookup result once it matches a cached flow. If the flow ID is valid, the Scheduler sends the packet to the pipeline with the minimum load i.e. the smallest number of packets among all the queues. This helps balance the load among pipelines. Since the flow ID is not valid until the first packet of the flow enters the pipelines, such a scheduling guarantees that, when the subsequent packets of a flow exit the pipeline, the first packet of this flow already completes lookup, and the next-hop information for this flow is available. When a packet exits the pipelines, it uses its flow ID to lookup Outbound Flow Table. If it is the first packet of a flow, it validates its entry in Outbound Flow Table and updates the next-hop information. Otherwise, it directly retrieves the next-hop information from Outbound Flow Table.

The entries in Inbound Flow Table store the information of the flows that are being looked up. It exploits the inherent caching of the architecture. Hence, in contrast to traditional caching schemes [22] which suffer from large pipeline delay, our scheme benefits from deep pipelining. Inbound Flow Table may store some entries for the flows that have completed lookup as well. Any cache replacement policy can be used to update the Inbound Flow Table. In our experiments in Section 6, we use the LRU (Least Recently Used) algorithm as default.

## 5.3 Preserving Intra-flow Packet Order

In our architecture, all packets are required to go through a pipeline from the first stage, despite whether they match any flow. Thus, the queued packets cannot catch up with their preceding packets that are already in the pipelines. In other words, only the queued packets can go out of order. Thus the Scheduler can detect the intra-flow out-of-order packets when sending packets to queues. If an intra-flow out-of-order packet is detected, a task to exchange the payload between out-of-order packets is arranged. As it takes multiple clock cycles for a packet to go through the pipelines, there is enough time for payload exchange to be completed before the packet exits the pipeline. Thus, the intra-flow packet order can be preserved. We use following notations.

- $p$ denotes a packet, and $f$ denotes a flow.
- $f(p)$ denotes the flow that $p$ belongs to.
- $POS(p)$ denotes $p$'s position in the queue. $POS(p) = 1$ if $p$ is at the head of the queue. If $p$ exits the queue, $POS(p) = 0$.
- $LP(f)$ denotes the position of the latest packet of the flow $f$.

**Algorithm 3** Lookup procedure based on flow pre-caching

**Input:** An arriving packet $p$.
**Output:** Its next-hop information.

```
1:  Compare with the flow entries in Inbound Flow Table.
2:  if No match then
3:      // Cache miss: p is the first packet of a flow.
4:      Assign a flow ID and add a new flow entry.
5:      Forward to the pipeline whose ID is obtained by in-
        dexing DITs.
6:      Go through the queue and the pipeline. When enter-
        ing the pipeline, validate its flow ID.
7:      Retrieve its next-hop information from the pipeline.
        Validate its entry in the Outbound Flow Table.
8:  else
9:      Assign the same flow ID as the matched flow.
10:     if The flow ID is valid then
11:         Forward to the pipeline whose queue has fewest
            packets.
12:         Exchange the payload to maintain the intra-flow
            packet order, if necessary.
13:     else
14:         Forward to the pipeline whose queue has the
            fewest packets among those that have more than
            LP(f(p)) packets.
15:     end if
16:     Go through the queue and pipeline. Skip any lookup
        operation.
17:     Retrieve next-hop information from Outbound Flow
        Table.
18: end if
```

### 5.3.1 Detecting Out-of-order Packets

The Inbound Flow Table records the position of the latest packet of each flow present in the architecture. As shown in Figure 7, if an arriving packet $p$ matches a flow $f$, the Scheduler sends $p$ to the pipeline with the minimum load. The Scheduler can detect whether $p$ goes out of order by comparing $POS(p)$ with $LP(f)$. If $POS(p) < LP(f)$, $p$ goes out of order within $f$. Then a task to exchange the payload of $p$ with that of the packet at $LP(f)$ is arranged. Meanwhile, $LP(f)$ is updated as $LP'(f) = \max[LP(f), POS(p)] - 1$.

### 5.3.2 Payload Exchange

When a packet enters the architecture, only its destination IP address is used for lookup. The payload (i.e. the entire packet) is stored in a buffer. The pointer to the payload in the buffer goes through the lookup engine, along with the IP address. To exchange the payload between two packets, we only need to exchange their pointers to the payload. The two packets that go out of order are both in some queues.

**Table 3. IP header traces**

| Trace | Location | Date | # of packets | # of unique IPs |
|---|---|---|---|---|
| APTH-I: AMP-1110523221-1 | Miami, USA | 20050311 | 769100 | 17628 |
| AUCK-VIII: 20031215-230000 | Auckland, New Zealand | 20031215 | 1449061 | 67865 |
| CNIC-I: SVL-LAX-20050319-110000-0 | Los Angeles, USA | 20050319 | 2104383 | 7275 |
| IPLS-IV: I2A-1091235138-1 | Atlanta, USA | 20040731 | 1821364 | 15791 |

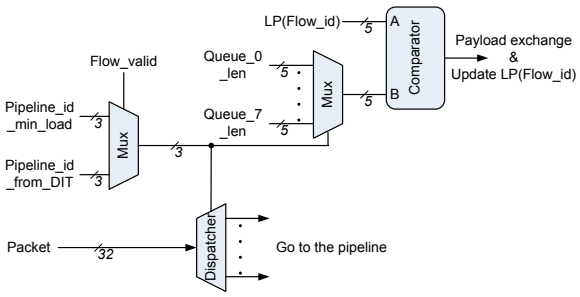


**Figure 7. Scheduler for one input.**

Thus, as the packets need several clock cycles to complete lookup in the pipelines, the process of payload exchange can be completed before the packets exit the pipelines.

The complete flow pre-caching procedure is shown in Algorithm 3.

# 6 Performance Evaluation

This section evaluates the effectiveness of the proposed schemes for balancing the traffic and preserving the intra-flow packet order. All experiments are based on simulation using real-life traffic traces.

## 6.1 Simulation Setup

Due to unavailability of public IP traces associated with their corresponding routing tables, we generated routing tables based on given traffic traces. We downloaded four anonymized real-life traffic traces from [16]. Their information is listed in Table 3. We extracted the unique destination IP addresses from the traces to build their routing tables. Details are shown in Table 3. According to the last two columns in Table 3, those traces presented vastly different locality on the destination IP distribution in the traffic.

The default setting of the major parameters of the proposed architecture is shown in Table 4, where the queue size is the maximum number of packets allowed in a queue, and the flow table size is the maximum number of flow entries in either Inbound Flow Table or Outbound Flow Table.

**Table 4. Parameter Settings**

| Parameter | Notation | Default value |
|---|---|---|
| # of pipelines | $P$ | 8 |
| Pipeline depth | $H$ | 25 |
| Queue size | $Q$ | 16 |
| Flow table size | $F$ | 200 |

## 6.2 Methodology

### 6.2.1 Pre-Caching vs. Caching

In the traditional caching schemes, caches are updated after the packets that have cache miss complete their lookup. Our architecture can easily implement the traditional caching schemes by skipping Outbound Flow Table and updating Inbound Flow Table only after the flows retrieve the next-hop information from the pipeline.

We developed a trace-driven simulator for both caching based and flow pre-caching based multi-pipeline architectures. In the following figures, "caching" refers to the traditional caching scheme, while "pre-caching" refers to the proposed flow pre-caching scheme.

### 6.2.2 Performance Metrics

The arrival rate of input packets in a $P$-pipeline architecture was $P$ per clock cycle. We measured four performance metrics: *Throughput*, *Packet drop rate*, *Latency*, and *Payload exchange frequency*.

The *Throughput* is defined as the average number of output packets per clock cycle. Note that in a $P$ pipeline architecture, the *Throughput* $\leq P$.

The *Packet drop rate* is defined as the ratio of the number of dropped packets to the total number of packets input.

The *Latency* is defined as the number of clock cycles taken for a packet to go through the queue and the pipeline.

The *Payload exchange frequency* is defined as the ratio of the number of times the payload is exchanged to the total number of packets processed.

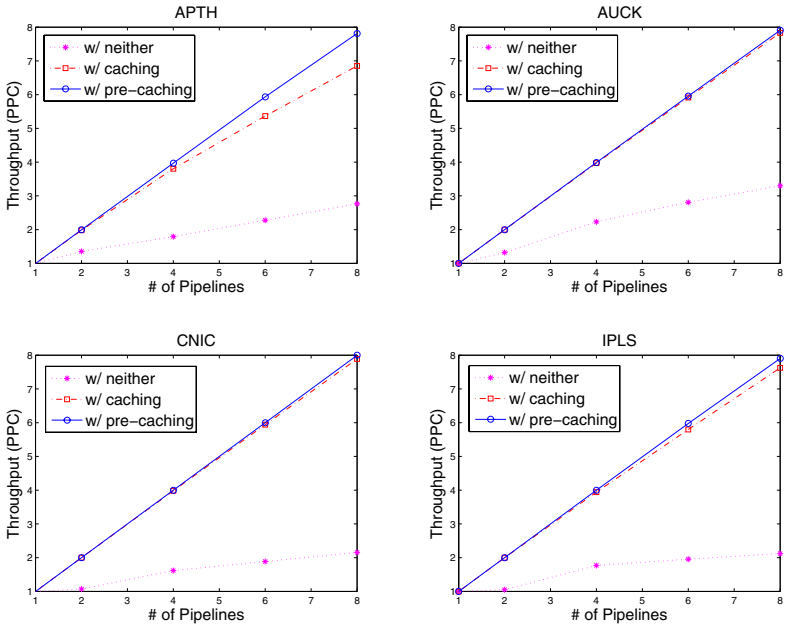


**Figure 8. Throughput with various numbers of pipelines ($P = 1, 2, 4, 6, 8$; $H = 25, Q = 16, F = 200$).**

### 6.3 Throughput Scalability

In this experiment, we increased $P$ and examined its effect on the throughput performance. The results are shown in Figure 8. According to the results, when neither caching nor pre-caching was enabled, the increase in the throughput exhibited poor scalability. The pre-caching scheme had good scalability in all cases. Especially for the *APTH* trace, the pre-caching scheme achieved a throughput of over 7.8 packets per clock cycle (PPC), while the caching scheme achieved no more than 6.8 PPC.

### 6.4 Increasing Pipeline Depth

We varied the number of pipeline stages to understand its impact on the packet drop rate. The results are shown in Figures 9. Deep pipelining had an adverse effect on the traditional caching scheme: the packet drop raised when the pipeline got deeper. This confirmed our discussion in Section 5.1. On the other hand, the proposed pre-caching scheme kept the packet drop rate to be a constant below 2.5 %, regardless of how deep the pipeline was.

### 6.5 Increasing Queue Size

Large queues can alleviate the access conflict when multiple packets are directed to the same pipeline at the same time. However, large queues also add additional delay

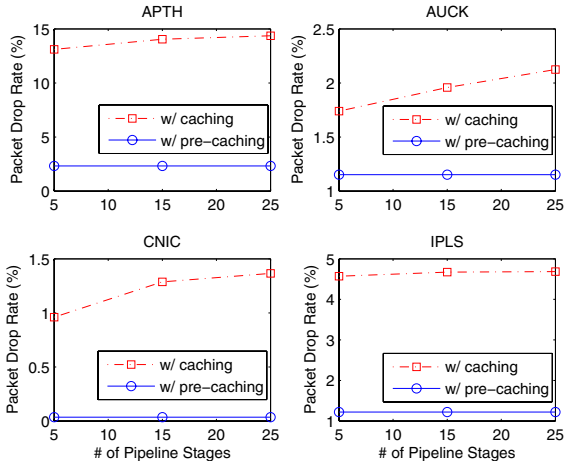


**Figure 9. Packet drop rate with increasing pipeline depth ($H = 5, 15, 25$; $P = 8, Q = 16, F = 200$).**

in processing packets. As discussed in Section 5.1, this may have adverse effect on the performance of traditional caching-based schemes. In this experiment, we varied the queue size and evaluated the packet drop rate. The results are shown in Figure 10. For the traces of AUCK and CNIC, the packet drop rate reduced when the queue size was increased. However, for APTH and IPLS, the packet drop

rate increased slightly in caching-based scheme, when the queue size was increased above 16. On the other hand, in the pre-caching based scheme, the packet drop rate for all traces reduced when the queue size was increased.

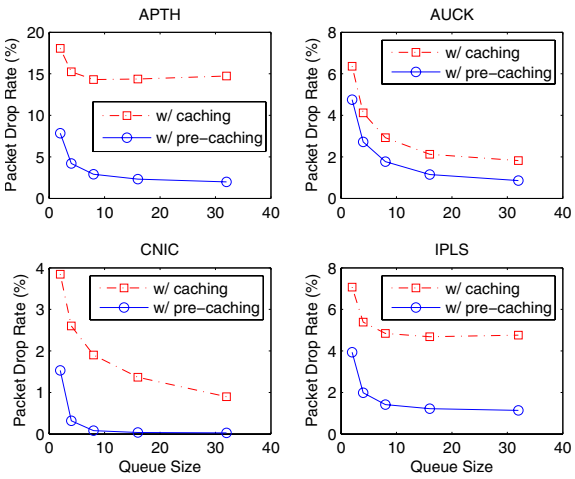


**Figure 10. Packet drop rate with varying queue size ($Q = 2, 4, 8, 16, 32$; $P = 8, H = 25, F = 200$).**

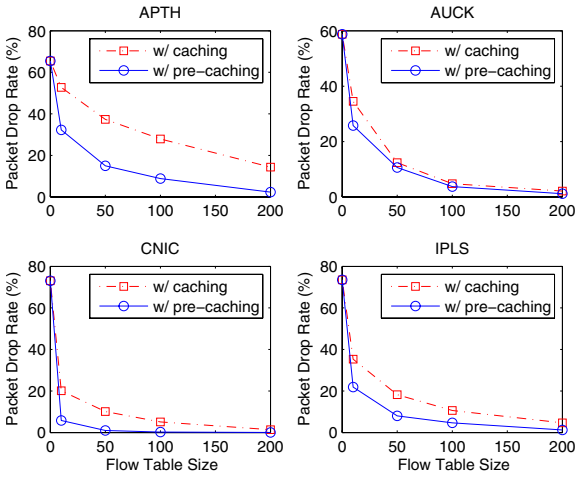


**Figure 11. Packet drop rate with varying flow table size ($F = 0, 10, 50, 100, 200$; $P = 8, H = 25, Q = 16$).**

### 6.6 Increasing Flow Table Size

A larger flow table can result in a lower cache miss rate. We varied the flow table size and obtained the results shown in Figure 11. As expected, when the flow table size was increased, the packet drop rate reduced, in both the schemes.

### 6.7 Latency Analysis

Queuing adds a variable delay on processing the packets. The number of clock cycles for a packet to go through the queue and the pipeline is between $1 + H$ and $Q + H$. We recorded the processing delay for each packet in the 8-pipeline architecture ($H = 25, Q = 16$). The results are shown in Figure 12. In each figure, we show the maximum and the minimum delay in two dotted blue lines respectively. As expected, the maximum delay was $16 + 25 = 41$ clock cycles, and the minimum delay was $1 + 25 = 26$ clock cycles. We also show the value of the average delay in a dashed red line. The average delay was around 30 clock cycles, for all traces.

### 6.8 Overhead Estimation

In all experiments, the intra-flow packet order was preserved due to payload exchange. However, high payload exchange frequency may incur high overhead in real implementations. We varied the number of pipelines to measure the payload exchange frequency. The results are shown in Figure 13. When $P$ was increased from 1 to 8, the payload exchange frequency for most traces increased mildly. Even for the *APTH* trace where the payload exchange frequency increased dramatically, the payload exchange frequency was still less than 10 %.

### 6.9 Overall Performance

According to Section 4.3.5, we can achieve a global clock rate of over 1.33 GHz. Meanwhile, our 8-pipeline architecture can achieve a throughput of over 7.8 PPC, as shown in Figure 8. The overall throughput of the proposed 8-pipeline architecture can be over 10 billion packets per second (GPPS), i.e. 3.2 Tbps for the packets with the minimum size of 40 bytes.

## 7 Conclusions

This paper proposed a parallel SRAM-based multi-pipeline architecture for terabit trie-based IP lookup. Memory and traffic balancing, and intra-flow packet ordering were identified as three major problems. We proposed a two-level mapping scheme to balance the memory requirement among the pipelines as well as across the stages in a pipeline. We proposed a flow pre-caching scheme to balance the traffic among multiple pipelines. It exploits the inherent caching of the architecture and benefits from deep pipelining. Also, a scheme called payload exchange was used to maintain the intra-flow packet order. Experimental

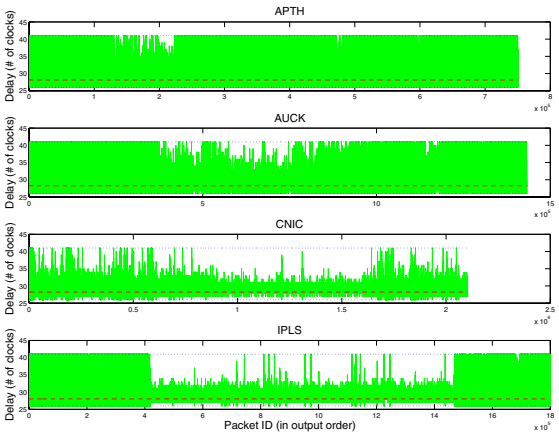


**Figure 12. Processing delay of each output packet ($P = 8, H = 25, Q = 16, F = 200$).**

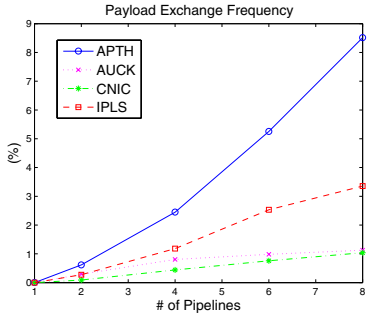


**Figure 13. Payload exchange frequency with varying numbers of pipelines ($P = 1, 2, 4, 6, 8; H = 25, Q = 16, F = 200$).**

results using real-life traffic traces show that the proposed architecture with 8 pipelines can achieve a high throughput of 3.2 Tbps.

Our recent work [10] further extends the proposed architecture. By replacing the single-port SRAMs with dual-port SRAMs, we allow each pipeline to be traversed from two directions at the same time. Our future work includes applying the SRAM-based pipeline architectures to multi-dimensional packet classification and deep packet inspection.

## 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, S. Rajgopal, L. Huang, and N. Richardson. Hardware implementation of a tree based IP lookup algorithm for OC-768 and beyond. In *Proc. DesignCon '05*.

[3] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. In *Proc. ISCA '05*, pages 123–133.

[4] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. In *Proc. INFOCOM '03*, pages 64–74.

[5] CACTI. http://www.hpl.hp.com/personal/norman_jouppi/cacti4.html.

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

[7] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34(2):97–122, 2004.

[8] S. Govind, R. Govindarajan, and J. Kuri. Packet reordering in network processors. In *Proc. IPDPS '07*, pages 1–10.

[9] W. Jiang and V. K. Prasanna. A memory-balanced linear pipeline architecture for trie-based IP lookup. In *Proc. HotI '07*, 2007.

[10] W. Jiang and V. K. Prasanna. Multi-terabit IP lookup using parallel bidirectional pipelines. In *Proc. CF '08*, 2008. To appear.

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

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

[13] S. Kumar, M. Becchi, P. Crowley, and J. Turner. CAMP: fast and efficient IP lookup architecture. In *Proc. ANCS '06*, pages 51–60.

[14] D. Lin, Y. Zhang, C. Hu, B. Liu, X. Zhang, and D. Pao. Route table partitioning and load balancing for parallel searching with TCAMs. In *Proc. IPDPS '07*, pages 1–10.

[15] W. Lu and S. Sahni. Packet forwarding using pipelined multibit tries. In *Proc. ISCC '06*, pages 802–807.

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

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

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

[19] 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.

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

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

[22] Y. Tung and H. Che. Study of flow caching for layer-4 switching. In *Proc. ICCCN '00*, pages 135–140.

[23] B. Wu, Y. Xu, H. Lu, and B. Liu. A practical packet reordering mechanism with flow granularity for parallelism exploiting in network processors. In *Proc. IPDPS '05*, pages 133a–133a.

[24] F. Zane, G. J. Narlikar, and A. Basu. CoolCAMs: Power-efficient TCAMs for forwarding engines. In *Proc. INFOCOM '03*, pages 42–52.

[25] 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.