A SRAM-based Architecture for Trie-based IP Lookup Using FPGA

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

Abstract

Internet Protocol (IP) lookup in routers can be implemented by some form of tree traversal. Pipelining can dramatically improve the search throughput. However, it results in unbalanced memory allocation over the pipeline stages. This has been identified as a major challenge for pipelined solutions. In this paper, an IP lookup rate of 325 MLPS (millions lookups per second) is achieved using a novel SRAM-based bidirectional optimized linear pipeline architecture on Field Programmable Gate Array, named BiOLP, for tree-based search engines in IP routers. BiOLP can also achieve a perfectly balanced memory distribution over the pipeline stages. Moreover, by employing caching to exploit the Internet traffic locality, BiOLP can achieve a high throughput of up to 1.3 GLPS (billion lookups per second). It also maintains packet input order, and supports route updates without blocking subsequent incoming packets.

Keywords: IP Address Lookup, Longest Prefix Matching, Reconfigurable Hardware, Field Programmable Gate Array (FPGA).

1. Introduction

With the rapid growth of the Internet, design of high speed IP routers has been a major area of research. Advances in optical networking technology are pushing link rates in high speed IP routers beyond OC-768 (40 Gbps). Such high rates demand that packet forwarding in IP routers must be performed in hardware. For instance, 40 Gbps links require a throughput of 8 ns per packet, i.e. 125 million packets per second (MPPS), for a minimum size (40 bytes) packet. Such throughput is impossible using existing software-based solutions [1], [2].

Supported by the United States National Science Foundation under grant No.CCR-0702784.

Most hardware-based solutions for high speed packet forwarding in routers fall into two main categories: ternary content addressable memory (TCAM)-based and dynamic/static random access memory (DRAM/SRAM)-based solutions. Although TCAM-based engines can retrieve 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 [3]. As shown in Table 1, SRAMs outperform TCAMs with respect to speed, density, and power consumption.

Table 1: Comparison of TCAM and SRAM technologies

	TCAM (18Mb chip)	SRAM (18Mb chip)
Maximum clock rate (MHz)	266 [4]	400 [5], [6]
# of transistors per bit [7]	16	6
Power consumption (Watts)	12 ~ 15 [8]	≈ 0.1 [9]

Since SRAM-based solutions utilize some kind of tree traversal, they require multiple cycles to perform a single lookup. Several researchers have explored pipelining to improve the throughput. A simple pipelining approach is to map each tree level onto a pipeline stage with its own memory and processing logic. One packet can be processed every clock cycle. However, this approach results in unbalanced tree node distribution over the pipeline stages. This has been identified as a dominant issue for pipelined architectures [10]. In an unbalanced pipeline, the "fattest" stage, which stores the largest number of tree 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 tree nodes stored in that stage. Particularly during the update process caused by intensive route/rule insertion, the fattest stage may also result in memory overflow. Furthermore, since it is unclear at hardware

978-0-7695-3307-0/08 $25.00 © 2008 IEEE
DOI 10.1109/FCCM.2008.9

design time which stage will be the fattest, we need to allocate memory with the maximum size for every stage. This over-provision results in memory wastage [11].

To balance the memory distribution across stages, several novel pipeline architectures have been proposed [11-13]. However, none of them can achieve a perfectly balanced memory distribution over stages. Moreover, some of them use non-linear structures and result in throughput degradation.

The key issues for any new architecture on IP lookup engine are: high throughput, maximal size of supported routing table, incremental update, in-order packet output, and power consumption. To address these challenges, we propose and implement a SRAM-based bidirectional optimized linear pipeline architecture, named BiOLP, for tree-based IP Lookup engine routers on FPGA. This paper makes the following contributions:

- To the best of our knowledge, this architecture is the first trie-based design that uses the on-chip FPGA resources only to support a fairly large routing table, Mae-West (rrc08, 20040901) [14].
- This is also the first architecture that employs IP caching on FPGA without using TCAM to effectively exploit the Internet traffic locality. A high throughput of more than 1 packet per clock cycle is obtained.
- The implementation results show the throughput of 325 MLPS for non-cache-based design. To the best of our knowledge, this is the fastest IP Lookup engine on FPGA; and 1.3 GLPS for cache-based design. This is a promising solution for the next generation IP routers.

The rest of the paper is organized as follows: Section 2 covers the background and related work; Section 3 introduces the BiOLP architecture; Section 4 describes BiOLP implementation; Section 5 presents implementation results; and Section 6 concludes the paper.

2. Background and Related Work

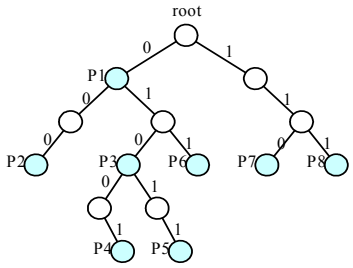
2.1. Trie-based IP Lookup

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 [1]. 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 for making 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 [15], 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). For simplicity, we consider only the leaf-pushed uni-bit trie in this paper, though our ideas also apply to other forms of tries [16].

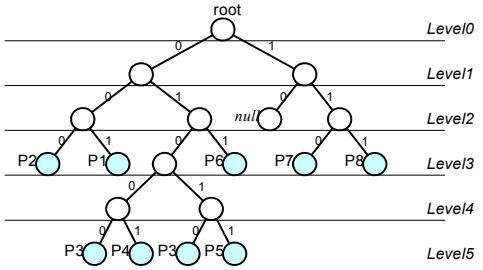
Given a leaf-pushed uni-bit trie, IP lookup is performed by traversing the trie according to the bits in the IP address. When a leaf is reached, the prefix associated with the leaf is the longest matched prefix for that IP address.

0*	P1
000*	P2
010*	P3
01001*	P4
01011*	P5
011*	P6
110*	P7
111*	P8

(a) Prefix set



(b) Uni-bit trie



(c) Leaf-pushed uni-bit trie

Figure 1: Prefix set; Uni-bit trie; Leaf-pushed uni-bit trie

2.2. Related Work

Since this work is based entirely on FPGA, we only cover some related works that are in the same domain. As mentioned above, there are two types of architecture on FPGA: TCAM-based and SRAM-based. Each of them has its own advantages and disadvantages.

2.2.1. TCAM-based Architectures

TCAM is used in some architectures to simplify the complexity of the designs. However, it also reduces the clock speed, and increases the power consumption of the entire system. Song et al. [17] introduce an archi-

tecture called BV-TCAM, which combines the TCAM and the Bit Vector (BV) algorithm to effectively compress the data representations and boost throughput. This design can only handle lookup rate of about 30 MLPS.

Kasnavi et al. [18] propose a similar idea of cache-based IP address lookup architecture, which is comprised of a non-blocking Multizone Pipelined Cache (MPC) and of a hardware-supported IP routing lookup method. They use two-stage pipeline for a half-prefix/half-full address IP cache that results in lower activity than conventional caches. However, this design does not use FPGA.

2.2.2. SRAM-based Architectures

Baboescu et al. [11] propose a Ring pipeline architecture for tree-based search engines. 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 an almost balanced pipeline. Some subtrees must wrap around if their roots are mapped to the last several stages. Any incoming IP packet needs to lookup an index table to find its corresponding subtree's root, which is the starting point of that search. Since the subtrees may be from different depths of the original tree, we cannot use a constant number of address bits to index the table. Thus, the index table must be built by content addressable memories (CAMs), which may result in lower speed. 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.

The fastest IP lookup implementation on FPGA to date is the architecture from Hamid et al. [19], which can achieve a lookup rate of 263 MLPS. Their architecture takes advantages of both traditional hashing scheme and reconfigurable hardware by implementing only the colliding prefixes (prefixes that have the same hashing value) on reconfigurable hardware, and other prefixes in a main table on memory. This architecture can support the Mae-West – rrc08 routing table, and can be updated using partial reconfiguration when adding or removing prefixes. However, it does not support incremental update, and the update time is bounded by the time in partial reconfiguration. Moreover, the power consumption of this design is potentially high due to a large amount of logic resources utilized.

Sangireddy et al. [20] propose two algorithms, Elevator-Stairs and log W-Elevators, which is scalable and memory efficient. However, their designs can only achieve up to 21.41 MLPS. Meribout et al. [21] present another architecture with the lookup speed of 66 MLPS. In this design, a commodity Random Access Memory (RAM) is needed in their design; and the achieved lookup rate is reasonably low.

As mentioned above, pipelining can dramatically improve the throughput of tree traversal. A straight forward way to pipeline a tree is to assign each tree level to a different stage, so that a packet can be processed every clock cycle. However, this simple pipeline scheme results in unbalanced memory distribution, leading to low throughput and inefficient memory allocation.

Kumar et al. [12] extend the circular pipeline with a new architecture called Circular, Adaptive and Monotonic Pipeline (CAMP). It uses several initial bits (i.e. initial stride) as the hashing index to partition the tree. Using the similar idea but different mapping algorithm from Ring [11], CAMP creates an almost balanced pipeline as well. Unlike the Ring pipeline, CAMP has multiple entry stages and exit stages. To manage the access conflicts between packets from current and preceding stages, several queues are employed. 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, some packets may be discarded [12]. In other words, the worst-case throughput is 0.8 packets per clock. Also in CAMP, a queue adds extra delay for each packet, which may result in an out-of-order output and delay variation.

3. BiOLP Architecture

3.1. The Overview

The block diagram of basic architecture and a basic stage are shown in Figure 2 and Figure 3, respectively. There is one *Direction Index Table* (DIT), which stores the relationship between the subtries and their mapping directions. For any arriving packet p, the initial bits of its IP address are used to lookup the DIT and retrieve the information about its corresponding subtrie *ST(p)*. The information includes (1) the distance to the stage where the root of *ST(p)* is stored, (2) the memory address of the root of *ST(p)* in that stage, and (3) the mapping direction of *ST(p)* which leads the packet to different entrance of the pipeline. For example, in Figure 2, if the mapping direction is top-down, the packet is sent to the leftmost stage of the pipeline. Otherwise, the packet is sent to the rightmost stage.

Once its direction is known, the packet will go through the entire pipeline in that direction. The pipeline is configured as a dual-entrance bidirectional linear pipeline. At each pipeline stage, the memory has dual

Read/Write ports so that the packets from both directions can access the memory simultaneously. The content of each entry in the memory includes (1) the memory address of the child node and (2) the distance to the stage where the child node is stored. Before a packet moves onto the next stage, its distance value is checked. If it is zero, the memory address of its child node will be 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.

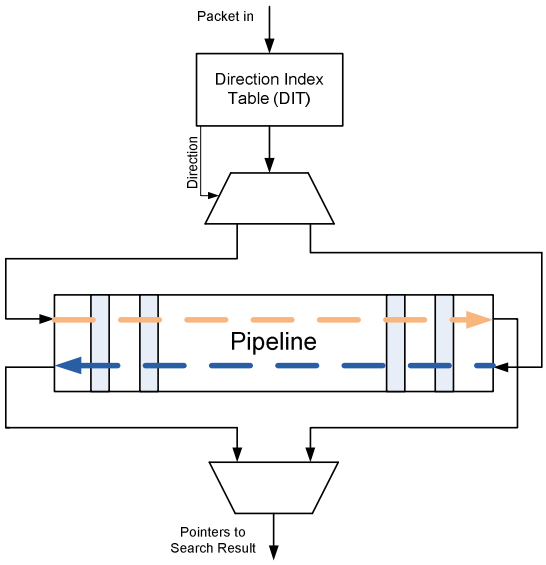


Figure 2: Block diagram of the basic architecture

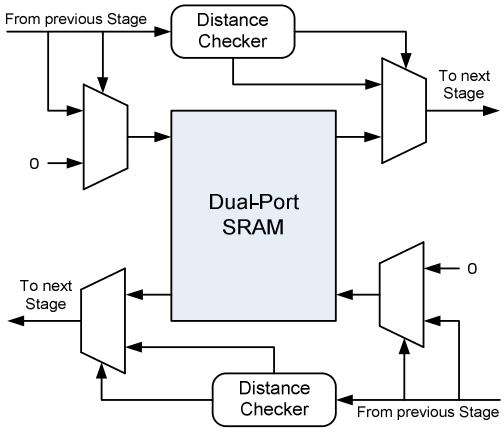


Figure 3: Block diagram of a basic pipeline stage

3.2. Memory Balancing

State-of-the-art techniques cannot achieve perfectly balanced memory distribution, due to several constraints they place during mapping:

- They require the tree nodes on the same level be mapped onto the same stage.
- The mapping scheme is uni-directional: the subtrees partitioned from the original tree must be mapped in the same direction (either from the root, or from the leaves).

However, both constraints are unnecessary. The only constraint we must obey is:

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

BiOLP employs dual-port on chip Block RAMs, allowing two flows from opposite directions to access the local memory in a stage at the same time. One may argue that dual-ported memory requires twice the access time compared with a single-ported one, leading to the reduction in clock speed of the design. However, the block-RAM offers dual-port feature; therefore, it does not make any difference when we configure it dual or single port. With a bidirectional fine-grained mapping scheme, a perfectly balanced memory distribution over pipeline stages is achieved. BiOLP has many desirable properties due to its linear structure:

- Each packet has a constant delay to go through the architecture.
- The packet input order is maintained
- Non-blocking update is supported, that is, while a write bubble is inserted to update the stages, both the subsequent and the antecedent packets can perform the search as well.

The main ideas of the proposed bidirectional fine-grained mapping scheme are to allow:

- Two subtries to be mapped onto the pipeline in two different directions, and
- Two trie nodes on the same trie level to be mapped onto different stages.

Input: K subtries.
Output: V subtries to be inverted

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

Figure 4: Algorithm for selecting the subtrie to be inverted

3.2.1. Subtrie Invertion

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:

Input: K forward subtries, V reverse subtries.
Output: H stages with mapped nodes.

1: Create and initialize two lists: $ReadyList = \emptyset$ and $NextReadyList = \emptyset$
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 $(Mi < \lceil R_n/R_e \rceil$ and $Readylist \neq \emptyset)$ **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 $N_c \geq R_h - 1$ **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**

Figure 5: Bidirectional fine-grained mapping algorithm

- *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.
- *Least height*: The subtrie whose height is the minimum is preferred. Due to *Constraint 1*, a subtrie with larger height has less flexibility to be mapped onto pipeline stages.
- *Largest leaf per height*: This is the combination of the previous two heuristics, by dividing the number of leaves of a subtrie by its height.
- *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.

Figure 4 shows the algorithm to find 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.

3.2.2. 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 shown in Figure 5. In this algorithm, M_i is the number of nodes mapped onto stage i. The node whose priority is equal to the number of the remaining stages is regarded as a $Critical$ node. If such a node is not mapped onto the current stage, none of its descendants can be mapped later. 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 will be 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.

3.2.3. Cache-based BiOLP Architecture

The proposed architecture of SRAM-based BiOLP can be used as an engine in a cache-based architecture, as shown in Figure 6. This design takes advantages of the internet traffic locality due to the TCP mechanism and application characteristics [22]. It consists of five modules: cache, 4-1 FIFO, BiOLP, Delay, and 5-4 FIFO module. Notation *m-n* FIFO denotes a FIFO with *m* inputs and *n* outputs. At the front is the cache module, which takes in up to four IP addresses at a time. The most recently searched packets are cached. Any arriving packet accesses the cache first. If a cache hit occurs, the packet will skip traversing the pipeline. Otherwise, it needs to go through the pipeline. For IP lookup, only the destination IP of the packet is used to index the cache. The cache update will be 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 in the implementation.

We can insert multiple packets at one clock cycle as long as there are enough copies of the cache for those packets to access simultaneously. Hence, we can insert at most four packets during one clock cycle. Without

caching, the packet input order is maintained due to the linear architecture. However, with caching, the packet which has a cache hit will skip traversing the pipeline, and hence may go out of order. We add buffers to delay outputting the packets that have cache hit, as shown in **Figure 6** (Delay module). The length of the delay buffer is equal to the sum of the pipeline depth and the queue length. By using these buffers, the packet input order can be preserved. Packets coming out of Delay module and BiOLP are buffered in 5-4 output FIFO.

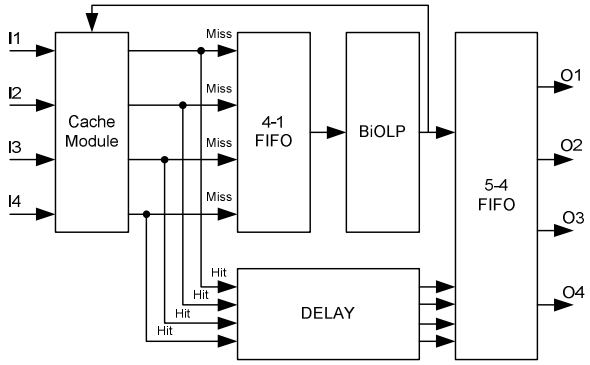


Figure 6: Top level block diagram of cache-based BiOLP

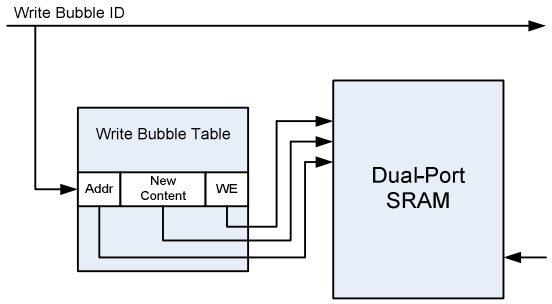


Figure 7: Route update using route bubbles

3.2.4. Route Update

We update the memory in the pipeline by inserting write bubbles [10]. The new content of the memory is computed offline. When an update is initiated, a write bubble is inserted into the pipeline. The direction of write bubble insertion is determined by the direction of the subtrie that the write bubble is going to update. Each write bubble is assigned an ID. As shown in Figure 7, there is one write bubble table (WBT) in each stage. It stores the update information associated with the write bubble ID. When it arrives at the stage prior to the stage to be updated, the write bubble uses its ID to lookup the WBT. Then it 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.

Since the subtrees mapped onto the two directions are disjoint, a write bubble inserted from one direction will not contaminate the memory content for the search from the other direction. Also, since the pipeline is linear, all packets preceding or following the write bubble can perform their searches while the write bubble performs an update. However, for major update that changes the structure of the search tree, the entire memory content of each pipeline stage and the direction index table need to be reloaded.

4. BiOLP Implementation

4.1. Cache Module

As mentioned above, fully associativity and LRU are used to implement the cache module. Our experiments show that 16-entry cache is sufficient for our purposes. Since this is a tree-based architecture, all the prefixes are stored in blockRAM (BRAM); it is desirable that a large amount of BRAM is used to store a maximum possible size of routing table. Therefore, this module is implemented based on caching algorithm shown in Figure 8, using only registers and logics. Each entry in the cache has a register to store packet's destination IP address, a register to store flow ID information, and a 4-bit saturated counter. This counter can be set to max (15), and decremented by 1 if its value is greater than 1.

Caching algorithm (LRU)
Hit counter and *written counter* are counters that belong to the *hit entry* and *written entry*, respectively.

1: All entries are initially reset to 0, max = # entries in cache -1
2: **repeat** for all incoming packets
3: **if** hit
4: Decrement all $counters$
5: Set hit counter to max
6: **else**
7: Decrement all $counters$ by 1
8: **if** there is $empty$ slot
9: Write new entry into empty slot
10: Set written counter to max
11: **else**
12: Pick the first slot with $counter = 1$
13: Write new entry into that slot
14: Set written counter to max
15: **end repeat**

Figure 8: Caching algorithm (LRU)

The caching algorithm ensures that there is always at least one slot in cache such that its counter has value of 0 or 1. This is because every time the algorithm sets one counter to max, it also decrements other counters by 1 (if their value is greater than 1). Therefore, there is no duplication of counter's value in the range from 2 to max. There are at most $max - 2$ values in this range, leaving at least 2 entries of value 0 or 1.

4.2. FIFO

There are two FIFOs in the architecture, 4-1 FIFO and 5-4 FIFO, as shown in Figure 6. Even though they have different numbers of input and output ports, their functionalities are identical. The 4-1 FIFO takes in up to 4 inputs and produces one output per clock cycle, while the 5-4 one takes in 5 inputs and produces 4 outputs. Since the number of inputs is different from the number of outputs in each FIFO, and there is only one clock for both writing and reading side, these are synchronous FIFOs with conversion. For simplicity, no handshaking feature is implemented, and therefore, any further arriving packet is dropped when the FIFO is full. Similar to the cache implementation, the two FIFOs are implemented using only registers and logics, to save BRAM for routing table. A common implementation of a FIFO is the circular buffer. A circular buffer is an array with read and write address. The read address is the index of the oldest queue entry, which will be the next to be removed; the write address points to the next free entry in the array (where the next value will be inserted). The size of the queue sets the maximum number of entries can be stored, and is determined by experiments.

4.3. BiOLP

BRAM is used to implement the direction index table (Figure 2), memory storage for all pipeline stages (Figure 3), and the write bubble table WBT (Figure 7). As mentioned before, we use the first 12 bits of an IP address as prefix expansion, and hence, the depth of direction index table is $\mathbf{2^{12}}$. Let $\boldsymbol{n}$ be the depth of memory storage in each pipeline stage. Each entry in DIT needs to store $\boldsymbol{n}$ address bits to index the first stage, plus one bit for direction, making the total of $\boldsymbol{n + 1}$ bits. 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 it 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. Since we have the total of 25 stages, we need 5 bits encoded as distance from the current lookup stage to the next one. Therefore, each entry (node) in the memory must include $\boldsymbol{n}$ bits for address, plus 5-bit for distance, and $\boldsymbol{m}$ bits for flow ID, or the total of $(\boldsymbol{n + 5 + m})$ bits. Our target chip, Virtex-4 FX140, has 9936 Kb of BRAM on chip, or 552 blocks of 18Kb, hence, each stage can have at most 22 blocks. Since we need to leave some blocks for the DIT and the WBT, 21 blocks is a good number for each stage, leaving 27 $(\mathbf{552 - 21 \times 25})$ blocks for DIT and write table.

5. Implementation Results

5.1. Memory Balancing

We conduct experiments on the four routing tables given in Table 2 to evaluate the effectiveness of the bidirectional fine-grained mapping scheme. In these experiments, the number of initial bits used for partitioning the trie is 12.

- ***Impact of the inversion heuristics***: As discussed in Section 3.2, we have four different *heuristics* to invert subtrees. Now we examine their performance and obtain the results. The value of the inversion factor is set to 1. According to the results, the *least average depth per leaf* heuristic has the best performance. Due to space limitation, only this result is illustrated in Figure 9. It shows that, when we have a choice, a balanced subtree should be inverted.
- ***Impact of the inversion factor***: In these experiments, we changed the value of the inversion factor. The inversion heuristic is the largest leaf heuristic. When the *inversion factor* is 0, the bidirectional mapping becomes top-down mapping only. The mapping turns to be bottom-up mapping when the inversion factor is large enough so that all subtrees are inverted. Hence, when we increase the inversion factor from 0 to 25, the bidirectional mapping changes from top-down to bottom-up. From the results, we can achieve perfect memory balance with the *inversion factor* between 4 and 8.

In summary, for trie-based IP lookup, the bidirectional fine-grained mapping scheme can achieve a perfectly balanced memory distribution over the pipeline stages, by either using an appropriate inversion heuristic or adopting an appropriate inversion factor.

Applying our mapping algorithm on different routing tables, we find that the largest routing table that our target FPGA chip can support is rrc08. With this

routing table, each stage has to store 14784 nodes, requiring 14-bit address line to index the memory. Let $m = 5 \Longrightarrow m + n + 5 = 24$, each stage is capable of handling at most 16128 entries, or nodes, and we can have up to 32 flows to forward packets to. The size of DIT can be calculated as $2^{12} \times (n + 1) = 15 \times 2^{12}$, or 4 blocks, leaving 23 blocks for the WBT. Each entry in the WBT has three fields: address (14 bits), content (24 bits), and WE (1 bit), for the total of 39 bits. With 23 available blocks, this WBT can store up to 10,000 entries. However, for simplicity, only 1 block is used in the implementation.

Table 2: Representative routing tables

Routing table	Date	# of prefixes	Max. prefix length	# of prefixes with length<16
rrc00	20040901	55875	32	739 (1.32%)
rrc01	20050101	41576	32	464 (1.12%)
rrc08	20060101	83662	24	495 (0.59%)
rrc11	20070412	154419	25	1150 (0.74%)

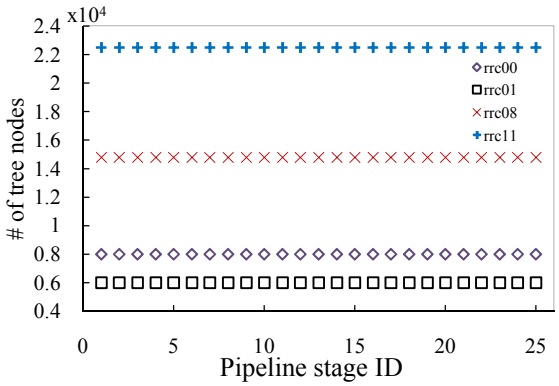


Figure 9: Bidirectional fine-grained mapping with "Least average depth per leaf" heuristic (Inversion factor = 1).

5.2. In-order Output

We define two notations of in-order output: *strict in-order* output (definition 1) and *flow-based in-order* output (definition 2). In the first definition, output packets strictly follow the order of the input packets, i.e. for any packet A and B, if packet A arrives before packet B, then A is output before B. In the later constraint, all packets that have the same destination IP address will have their arriving order reserved.

The non-cache-based design (BiOLP without cache) satisfies the strict constraint due to its linear pipeline and in-order input. However, the cache-based design can only guarantee the second constraint. Consider a flow of incoming packets with the same destination IP address, if the first packet misses in the cache, the subsequence packets are also missed until the cache is updated. Since we only update the cache at the end of the pipeline, and all the packets that have a cache hit are delayed by the depth of the pipeline plus the length of the input queue, the packets in the same flow will come out in-order.

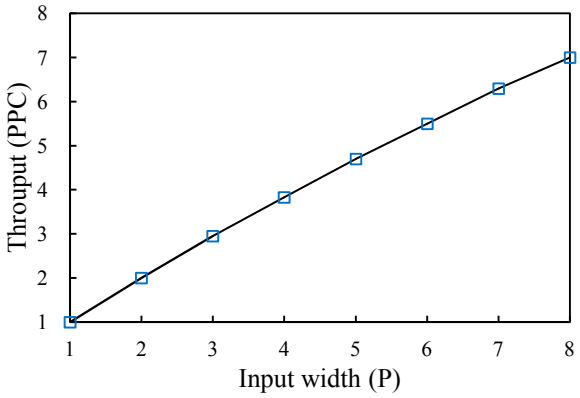


Figure 10: Throughput vs. Input width. ($\boldsymbol{H = 25, Q = 2, C = 160}$)

5.3. Throughput

The architecture was implemented in VHDL, using Xilinx ISE 9.1, and Virtex-4 FX140 as the target. The implementation results show a minimum clock period of 3.069 ns, or a maximum frequency of 325 MHz. Thus, the non-cache-based design can handle a lookup rate of 325 MLPS, or over 100Gbps data rate (with the minimum packet size of 40 bytes), which is 7.5 times the OC-256 rate.

We used real-life Internet traffic trace (APTH: AMP-1110523221-1, 769100 packets, 17628 unique entries in routing table [22]) to conduct experiments on the major parameters of the architecture to evaluate their impact on the throughput. These parameters include the input width, i.e. the number of parallel inputs, denoted P (input width); the pipeline depth, denoted H; the queue size, i.e. the maximum number of packets allowed to be stored in a queue, denoted Q; and the cache size, i.e. the maximum number of packets allowed to be cached, denoted C. In these experiments, the default settings for the architecture parameters are $P = 4, H = 25, Q = 2, C = 160$. The performance metric is the throughput in terms of the number of packets processed per clock cycle (PPC). Note that in P-width (P inputs) architecture, the throughput $\leq P$.

- ***Impact of the input width***: We increased the input width, and observed the throughput scalability. As shown in Figure 10, with caching, the throughput scales well with the input width.
- ***Impact of the cache size and the queue size***: We evaluated the impact of the cache size and the queue size, respectively, on the throughput. The

results are shown in Figure 11 and Figure 12. Caching is effective in improving the throughput. Even with only 1% of the routing entries being cached, the throughput reached almost 4 PPC in 4-width architecture. On the other hand, the queue size greater than 16 had little effect on the throughput improvement since we reach the point of diminishing return at the queue size of 16. Even without the queue, the throughput can reach over 3.65 PPC. Therefore, a small queue with the size of 16 is sufficient for the 4-width BiOLP architecture.

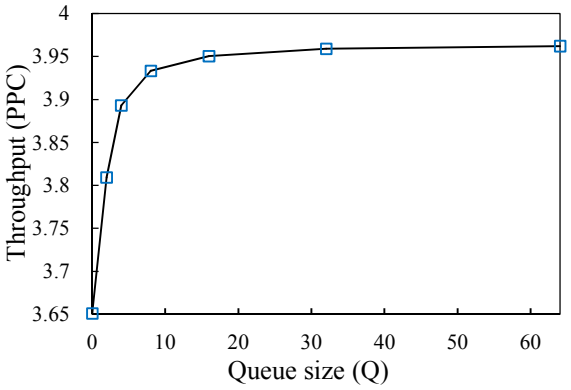


Figure 11: Throughput vs. Queue size. ($\boldsymbol{P = 4, C = 160}$)

Considering the throughput of 4 PPC, the overall throughput can be as high as $4 \times 325M = 1.3G$ packets per second, i.e 416 Gbps for the minimum packet size of 40 bytes, which is 2.6 times the OC-3072 rate. Such a throughput is also 144% higher than that of the state-of-the-art TCAM-based IP lookup engines [8].

We also conduct the same experiments with different real-life Internet traffic trace (IPLS: I2A-1091235138-1, 1821364 packets, 15791 unique entries in routing table [22]). The results are similar (actually slightly better) to those of APTH trace. However, due to the space limitation of this paper, the results are not presented.

5.4. Comparisons

Two key comparisons are performed with respect to memory balancing and throughput. The two candidates are: the Ring architecture [11] (architecture 1), and the state of the art architecture on FPGA [19] (architecture 2) since they can support the same routing table as our proposed architecture. All the resource values are normalized to Virtex-4 FX140, as shown in Table 3.

The Ring architecture can achieve almost but not perfectly balanced memory distribution over pipeline stages for IP lookup. The throughput of Ring is 1 packet per 2 clock cycles (i.e. 0.5 PPC), which can be increased to 1 PPC by using dual-port memories. This is still lower than the throughput of BiOLP. Since every packet needs to traverse the pipeline twice, the processing delay of Ring is twice that of BiOLP. Also, due to the delay, Ring needs a larger packet buffer for storing the packets in transit. Overall, the BiOLP outperforms Ring with respect to memory balancing and throughput (8×).

Table 3: Resource utilization and Throughput

Architecture	# slices (%)	BRAM	Throughput
1	1405(2.3%)	530	125 MLPS
2	14274(21%)	254	263 MLPS
Our (non-cache)	1785(3%)	530	325 MLPS
Our (cache)	4756(8%)	530	1.3 GLPS

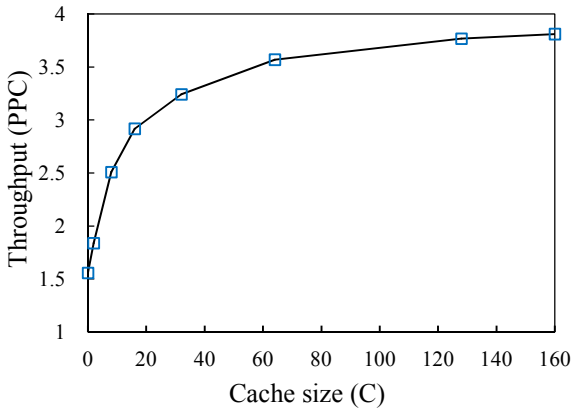


Figure 12: Throughput vs. Cache size. ($\boldsymbol{P = 4, Q = 2}$)

The architecture in [19], which was implemented on Virtex-2 Pro 100, can achieve up to 263 MLPS, utilizing 276 RAM blocks, and about 21% of the available slices (normalized to Virtex-4 FX140). Compared to our architecture, this uses half of the amount of BRAM, but 3 times the amount of available slices. Thus, the placing and routing is more complicated in their design, resulting in a lower clock speed. Our design outperforms their design by a factor of 4.94. Furthermore, our architecture supports static/dynamic RAM incremental update at run time without any CAD tool involvement, by inserting the write bubble whenever there is an update. In contrast, their design relies in partial reconfiguration, which requires modifying, re-synthesizing, and re-implementing the code to generate the partial bitstream. Our design also supports in-order output packets, and has lower power consumption.

6. Concluding Remarks

This paper proposed and implemented a SRAM-based bidirectional optimized linear pipeline architec-

ture, named BiOLP, for tree-based search engines in IP routers, without using external SRAM and TCAM. By using a bidirectional fine-grained mapping scheme, the tree nodes can be evenly distributed onto the pipeline stages. It results in a perfectly balanced memory allocation over the stages. As the result of this memory balancing, BiOLP can support a fairly large size of backbone routing table, Mae-West – rrc08. BiOLP also maintains the packet input order and supports non-blocking route update. It also employs packet caching to improve the throughput. BiOLP can achieve a high throughput of up to 416 Gbps i.e. 2.6 times the OC-3072 rate.

We plan to study more heuristics and compressing algorithms to support bigger routing table, i.e. rrc11, and IPv6. We also like to investigate the variable delay buffer to make the cache-based BiOLP support strictly in-order output packets. Our future work also includes implementing the BiOLP architecture on FPGAs for packet classification and evaluating its performance under real-life scenarios.

References

[1] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network,* vol. 15, no. 2, pp. 8-23, 2001.

[2] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network,* vol. 15, no. 2, pp. 24-32, 2001.

[3] 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*, Santa Clara, CA, 2005.

[4] Renesas CAM ASSP Series. [Online]. Available: http://www.renesas.com

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

[6] SAMSUNG High Speed SRAMs. [Online]. Available: http://www.samsung.com

[7] M. J. Akhbarizadeh, M. Nourani, D. S. Vijayasarathi, and P. T. Balsara, "A non-redundant ternary CAM circuit for network search engines," *IEEE Trans. VLSI Syst.,* vol. 14, no. 3, pp. 268-278, 2006.

[8] K. Zheng, C. Hu, H. Lu, and B. Liu, "A TCAM-based distributed parallel IP lookup scheme and performance analysis," *IEEE/ACM Trans. Netw.,* vol. 14, no. 4, pp. 863-875, 2006.

[9] CACTI. [Online]. Available: http://quid.hpl.hp.com:9081/cacti/

[10] A. Basu and G. Narlikar, "Fast incremental updates for pipelined forwarding engines," *IEEE/ACM Trans. Netw.,* vol. 13, no. 3, pp. 690-703, 2005.

[11] 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*, 2005.

[12] S. Kumar, M. Becchi, P. Crowley, and J. Turner, "CAMP: fast and efficient IP lookup architecture," in *Proc. ANCS '06*, San Jose, California, USA, 2006.

[13] W. Jiang and V. K. Prasanna, "A Memory-Balanced Linear Pipeline Architecture for Trie-based IP Lookup," in *Proc. HotI '07*, 2007.

[14] RIS Raw Data. [Online]. Available: http://data.ris.ripe.net

[15] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Trans. Comput. Syst.,* vol. 17, pp. 1-40, 1999.

[16] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: hardware/software IP lookups with incremental updates," *SIGCOMM Comput. Commun. Rev.,* vol. 34, no. 2, pp. 97-122, 2004.

[17] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays* Monterey, California, USA: ACM, 2005.

[18] S. Kasnavi, P. Berube, V. Gaudet, and J. N. Amaral, "A cache-based internet protocol address lookup architecture," *Comput. Networks,* vol. 52, no. 2, pp. 303-326, 2008.

[19] H. Fadishei, M. S. Zamani, and M. Sabaei, "A novel reconfigurable hardware architecture for IP address lookup," in *Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems* Princeton, NJ, USA: ACM, 2005.

[20] R. Sangireddy, N. Futamura, S. Aluru, and A. K. Somani, "Scalable, memory efficient, high-speed IP lookup algorithms," *IEEE/ACM Trans. Netw.,* vol. 13, no. 4, pp. 802-812, 2005.

[21] M. Meribout and M. Motomura, "A new hardware algorithm for fast IP routing targeting programmable routers," in *Network control and engineering for Qos, security and mobility II*: Kluwer Academic Publishers, 2003, pp. 164-179.

[22] NLANR network traffic packet header traces. [Online]. Available: http://pma.nlanr.net/Traces/