A Memory-Balanced Linear Pipeline Architecture for Trie-based IP Lookup

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

Rapid growth in network link rates poses a strong demand on high speed IP lookup engines. Trie-based architectures are natural candidates for pipelined implementation to provide high throughput. However, simply mapping a trie level onto a pipeline stage results in unbalanced memory distribution over different stages. To address this problem, several novel pipelined architectures have been proposed. But their non-linear pipeline structure results in some new performance issues such as throughput degradation and delay variation. In this paper, we propose a simple and effective linear pipeline architecture for trie-based IP lookup. Our architecture achieves evenly distributed memory while realizing high throughput of one lookup per clock cycle. It offers more freedom in mapping trie nodes to pipeline stages by supporting $nops$. We implement our design as well as the state-of-the-art solutions on a commodity FPGA and evaluate their performance. Post place and route results show that our design can achieve a throughput of 80 Gbps, up to twice the throughput of reference solutions. It has constant delay, maintains input order, and supports incremental route updates without disrupting the ongoing IP lookup operations.

1. Introduction

With the continuing growth of Internet traffic, IP address lookup has been a significant bottleneck for core routers. Advances in optical networking technology have pushed link rates in high speed routers beyond 40 Gbps, and Terabit links are expected in near future. To catch up with the rapid increase of link rates, IP lookup in high speed routers must be performed in hardware. For example, OC-768 (40 Gbps) links require a throughput of 8 ns per lookup for a minimum size (40 bytes) packet. Software-based solutions cannot support such rates.

Current hardware-based solutions for high speed IP lookup can be divided into two main categories: TCAM-based and SRAM-based solutions. Although TCAM-based engines can retrieve IP lookup results in just one clock, their throughput is limited by the low speed of TCAM[1]. SRAM outperforms TCAM with respect to speed, density and power consumption, but traditional SRAM-based engines need multiple clock cycles to finish a lookup. As pointed out by a number of researchers, using pipelining can significantly improve the throughput. For trie-based IP lookup, a simple approach is to map each trie level onto a private pipeline stage with its own memory and processing logic. With multiple stages in the pipeline, one IP packet can be looked up during a clock period. However, this approach results in unbalanced trie node distribution over different pipeline stages. This has been identified as a dominant issue for pipelined architectures [1, 2, 15]. In an unbalanced pipeline, the stage storing a larger number of trie nodes needs more time to access the larger memory. It also results in more frequent updates, which are proportional to the number of trie nodes stored in the local memory. When there is intensive route insertion, the larger stage can lead to memory overflow. Hence, such a heavily utilized stage can become a bottleneck and affect the overall performance of the pipeline.

To address these problems, some novel pipeline architectures have been proposed for implementation using ASIC technology. They achieve a relatively balanced memory distribution by using circular structures. However, their non-linear pipeline structures result in some new performance issues, such as throughput degradation and delay variation. Moreover, their performance is evaluated by estimation rather than on real hardware. For example, CACTI [3], a popular tool for estimating the SRAM performance has been used. However, such estimations do not consider many implementation issues, such as routing and logic delays. The actual throughput when implemented on FPGAs may be lower.

[1]Currently the highest advertised TCAM speed is 133 MHz while state of the art SRAMs can easily achieve clock rates of over 400 MHz.

1550-4794/07 $25.00 © 2007 IEEE
DOI 10.1109/HOTI.2007.10

In this paper, we focus on trie-based IP lookup engines that utilize pipelining. Linear pipeline architecture is adopted due to its desirable properities, such as constant delay and high throughput of one output per clock cycle. Using a fine-grained node-to-stage mapping, trie nodes are evenly distributed across most of the pipeline stages. For a realistic performance evaluation, we implement our design as well as the state-of-the-art solutions on a commodity FPGA. Post place and route results shows that, the proposed architecture can achieve a throughput of 80 Gbps for minimum size (40 bytes) packets on a single Xilinx Virtex II Pro FPGA [19]. Average memory usage per entry is 115.2 bits, excluding the next-hop information. In addition, our design supports fast incremental on-line updates without disruption to the ongoing IP lookup process.

The rest of the paper is organized as follows. In Section 2, we review the background and related works. In Section 3, we propose our optimized design named Optimized Linear Pipeline (OLP) architecture. In Section 4, we implement on FPGAs the OLP architecture as well as state-of-the-art pipelined architectures, and then compare their performance. Finally, in Section 5, we conclude the paper.

2. Background

IP lookup has been extensively studied [4, 13, 18]. From the perspective of data structures, these techniques can be classified into two main catergories: trie-based [8, 11, 14, 16] and hash-based solutions [5, 7]. In this paper, we consider only trie-based IP lookup which is naturally suitable for pipelining.

2.1 Trie-based IP Lookup

A trie is a tree-like data structure for longest prefix matching. Each prefix is represented by a node in the trie, and the value of the prefix corresponds to the path from the root of the tree to the node. The prefix bits are scanned left to right. If the scanned bit is 0, the node has a child to the left. A bit of 1 indicates a child to the right. The routing table in Figure 1 (a) corresponds to the 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).

IP lookup is performed by traversing the trie according to the bits in the IP address. When a leaf is reached, the last seen prefix along the path to the leaf is the longest matching prefix for the IP address. The time to look up a uni-bit trie (which is traversed in a bit-by-bit fashion), is equal to the prefix length. The use of multiple bits in one scan increases the search speed. Such a trie is called a multi-bit trie. The number of bits scanned at a time is called *stride*.

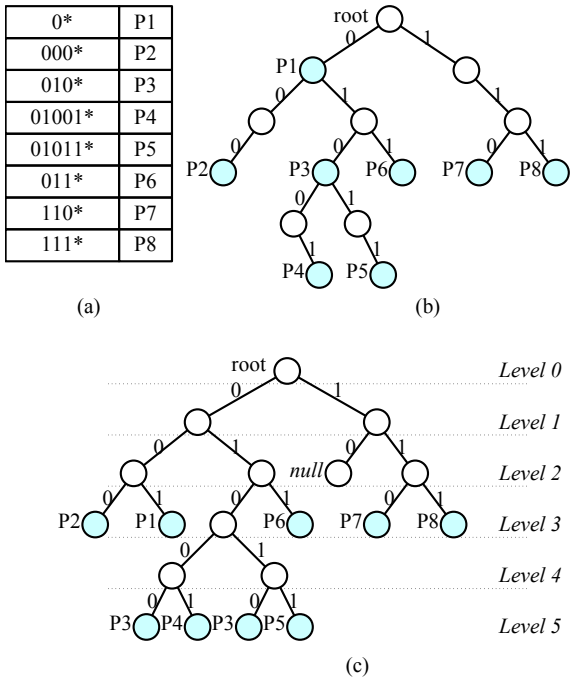

Figure 1. (a) Prefix set; (b) Uni-bit trie; (c) Leaf-pushed trie.

Normally each trie node contains two fields: the represented prefix and the pointer to the child nodes. By using an optimization called *leaf-pushing* [17], each node needs only one field: either the prefix index or the pointer to the child nodes. Some optimization schemes [4, 6] are also 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 can be applied to other more advanced tries.

2.2 Pipelined Architectures

A straightforward way to pipeline a trie is to assign each trie level to a distinct stage so that a lookup request can be issued every cycle, thus increasing the throughput. However, this simple pipeline scheme results in unbalanced memory distribution, leading to low throughput and inefficient memory allocation [1, 15].

Basu et al. [2] and Kim et al. [8] both reduce the memory imbalance by using variable strides to minimize the largest trie level. However, even with their schemes, the size of the memory of different stages can have a large variation. As an improvement upon [8], Lu et al. [10] proposes a tree-packing heuristic to further balance the memory, but it does not solve the fundamental problem of how to retrieve one node's descendents which are not allocated in the following

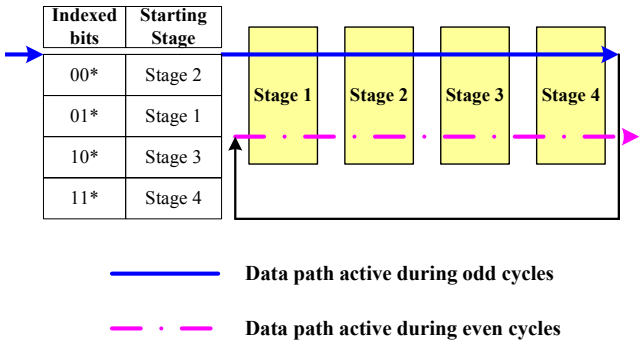

Figure 2. Example of Ring pipeline architecture [1].

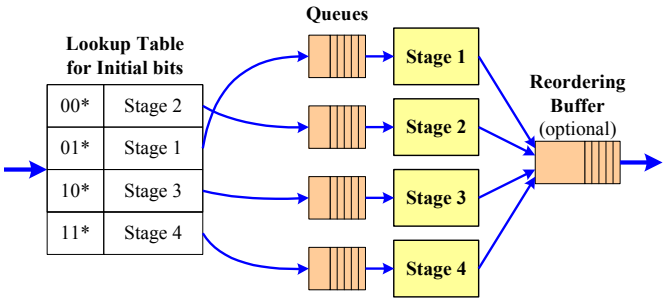

Figure 3. Example of CAMP architecture [9].

stage. Furthermore, a variable stride multi-bit trie is difficult for hardware implementation especially if incremental updating is needed [2].

Baboescu et al. [1] propose a circular pipelined trie, which is different from the previous approaches. The pipeline stages are configured in a circular, multi-point access pipeline so that lookup requests can be initiated at any stage (such a stage is called the starting stage or starting point for that lookup request). A trie is split into many small subtries of equal size. These subtries are then mapped to different stages to create a balanced pipeline. Some subtries have to 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 subtrie's root which is the starting point of that IP lookup request. Though all IP packets enter the pipeline from the first stage, their lookup processes may be activated at different stages. To prevent conflicts due to the arbitrary starting-stage, a Ring pipeline is proposed as shown in Figure 2. Each stage accommodates two data paths. The first path is used for the IP lookup request's first traversal of the pipeline. It is active during the odd clock cycles. The second path is active during the even clock cycles, allowing the request to continue its execution in the pipeline until it is finished. Hence, the throughput is 0.5 lookups per clock.

Kumar et al. [9] extend the circular pipeline with a new architecture called Circular, Adaptive and Monotonic Pipeline (CAMP), as shown in Figure 3. It uses several initial bits (i.e. initial stride) as the hashing index to partition the trie. Using the similar idea but different mapping algorithm from [1], CAMP creates a balanced pipeline as well. Unlike the Ring pipeline, CAMP has multiple entry stages and exit stages. The throughput may increase when the number of pipeline stages exceeds the subtries' height. To manage the access conflicts between requests from current and preceding stages, several *request queue*s are employed. Since different packets an input stream may have different entry and exit stages, the ordering of the packet stream is lost when passing through CAMP. Assuming the IP lookup requests traverse all the stages, when the IP lookup request arrival rate exceeds 0.8 packets per clock, some requests may be discarded [9]. In other words, the worst-case throughput is 0.8 lookups per clock.

In CAMP, a request queue results in extra delay for each request. The delay is defined as the number of clock cycles for a request measured between entering the request queue and exiting the pipeline. We conduct a software simulation to observe the worst-case delay for a 24-stage pipeline. The request queue size is 32 and the request arrival rate is 0.8. We assume the trie has 20 levels so that each IP lookup request exits the pipeline after it traverses 20 stages. To simulate the worst case, we always insert the new request into the queue whose corresponding stage is visited by the oldest request in the pipeline. Since the oldest request is followed by the most number of requests colliding with the new request, the delay for the new request to wait for entering the pipeline is maximized. Table 1 shows the average delay and the maximum delay for the worst cases. The burst length[2] of requests varies from 1 to 40. According to Table 1, the worst-case maximum request delay is 94 clock cycles. Considering that the minimum delay for a request is 20, we can conclude that, the delay for passing CAMP varies from 20 to 94 clock cycles under a traffic of 0.8 request arrivals per clock cycle. Such a large delay variation may have an adverse effect on some real-time Internet applications.

Multiple starting-points in either the Ring pipeline or CAMP do not support well the *write bubble* proposed in [2] for the incremental route update. For example, if the *write bubble* is assigned to start from some stage, it cannot enter the pipeline until no request in the pipeline needs to visit that stage. Hence, for both the Ring pipeline and CAMP, the pipeline operations need to be disrupted during the route update.

[2]The burst length is defined as the number of consecutive IP lookup requests assigned to the same pipeline stage.

Table 1. Worst-case delay versus request burst length

Burst Length	Average Delay	Maximum Delay
1	93.83	94
8	90.20	91
16	81.35	83
24	88.08	91
32	79.29	83
40	54.46	59

All the above problems force us to rethink the linear pipeline architecture which has some nice properties. For example, the linear pipeline can achieve a throughput of one lookup per clock cycle and can support *write bubble*s for incremental updates without disrupting the ongoing pipeline operations. The delay to go through a linear pipeline is always constant, equal to the number of pipeline stages.

3 Linear Pipeline with Balanced Memory

3.1 Motivation for OLP

Both the Ring pipeline and CAMP balance the trie node distribution by partitioning the trie into several subtries and then mapping them onto different pipeline stages. They both relax the constraint that all subtries' roots must be mapped to the first stage. The resulting pipelines have multiple starting-points for the lookup process. This causes access conflicts in the pipeline and reduces throughput. Hence, in our design, we enforce the constraint:

Constraint 1. *All the subtries' roots are mapped to the first stage.*

Thus, there is only one starting-point into the pipeline.

Both the Ring pipeline and CAMP have some restrictions on node-to-stage mapping. They both map a subtrie's nodes to pipeline stages in a level-by-level fashion, where nodes on the same level of a subtrie are mapped to the same pipeline stage. In contrast, our design offers more freedom to allow nodes on the same level of a subtrie to be mapped to different stages. For example, there are three subtries in Figure 4 (a) and node P1 and node P2 are on the same level of the second subtrie. As shown in Figure 4 (c), by inserting *nop*s, our scheme maps P2 to Stage 2 while it maps P1 to Stage 5, which makes the node distribution more uniform. The only constraint we must obey is:

Constraint 2. *If node A is an ancestor of node B in a subtrie, then A must be mapped to a stage preceding the stage B is mapped to.*

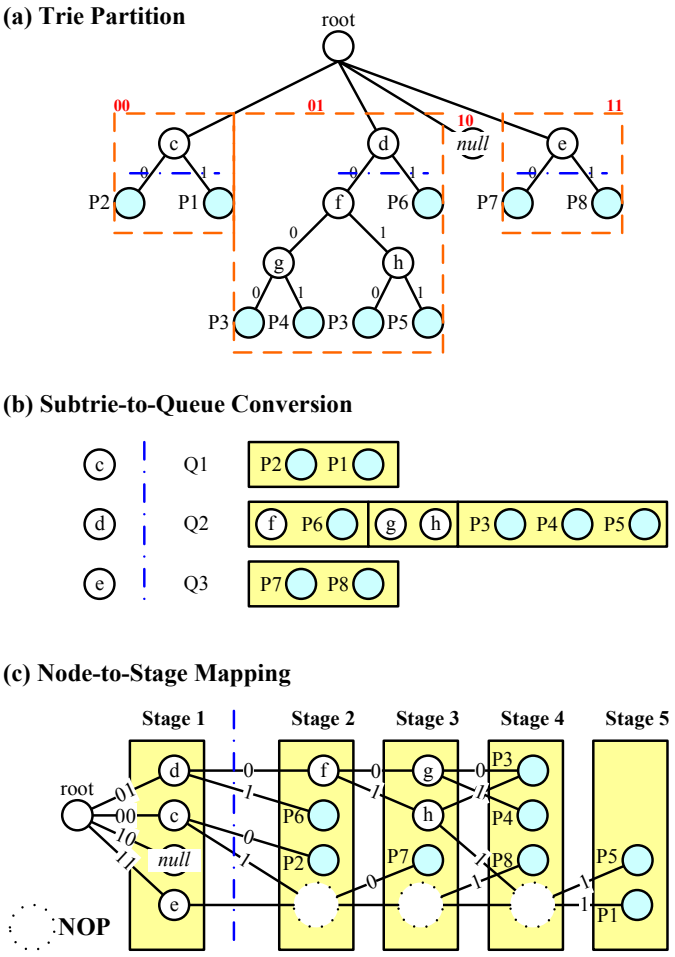

Figure 4. OLP architecture.

Based on the above discussion, we adopt the linear pipeline architecture, and use a fine-grained mapping scheme in a node-by-node fashion. Such an optimized linear pipeline (OLP) architecture can achieve the following objectives:

- throughput of one lookup per clock cycle;
- balanced memory requirement over pipeline stages;
- constant delay for all IP lookup requests.

3.2 Trie Partition

Similar to the Ring pipeline and CAMP, the first step is to partition the trie into multiple subtries. We use the same scheme as in [9], but with a different rule to determine the initial stride which is used to expand the prefixes. Since the first stage consists of all the subtries' roots, it cannot be balanced by moving nodes or inserting *nop*s. Adjusting the initial stride becomes the only way to balance the memory

requirement of the first stage with that of other stages.

Given a leaf-pushed uni-bit trie, we use prefix expansion to split the original trie (shown in Figure 1 (c)) into multiple subtries, as shown in Figure 4 (a). To describe the algorithm to determine the value of the initial stride, we have following notations.

I: value of the initial stride which is used for prefix expansion;

P: number of pipeline stages;

T: the original leaf-pushed uni-bit trie;

T': the new trie after using I to expand prefixes in T;

$N(t)$: total number of nodes in trie t;

$H(t)$: height of trie t. The height of a trie node is defined as the maximum distance from it to a leaf node. The height of a trie is defined as the height of the root of the trie.

Input: T, P
Output: I
$I \leftarrow \max[1, H(T) - P]$;
while *true* **do**
  use I to expand prefixes in T, and get T';
  if $2^{I-1} < \frac{N(T')}{P} < 2^I$ **then**
    return I;
  end
  $I \leftarrow I + 1$;
end

Algorithm 1: Determining the value of the initial stride.

Our goal is to make the number of subtries approximately equal to the average number of nodes over the pipeline stages. Also, the height of any subtrie must be less than P; otherwise such a subtrie cannot fit into the pipeline. Suppose we have $P = 5$ pipeline stages. We use the trie in Figure 1 (c) as an example. Its height is 5. We use $I = 1$ to expand the prefixes in that trie and obtain two subtries. According to Constraint 1, there are 2 nodes in the first stage while there are 16 nodes left to be assigned to the next 4 stages. The pipeline is unbalanced, since at least one stage has twice the number of nodes in the first stage. As shown in Figure 4 (a), if $I = 2$, we obtain three subtries and the average number of nodes over subsequent stages is $12/4 = 3$. Hence the algorithm will return $I = 2$ since it is more likely to balance the pipeline.

3.3 Mapping

As discussed above, under Constraint 2, OLP allows the nodes on the same level of a subtrie to be mapped onto different pipeline stages. It provides more flexibility to map the trie nodes to achieve a balanced memory. Assume there are S subtries after partitioning the original trie T, as described in Section 3.2. Assume the pipeline has P stages. We use a simple heuristic to perform the mapping, with the following notations.

ST_i: the ith subtrie, $i = 1, 2, \cdots, S$;

SG_i: the ith pipeline stage, $i = 1, 2, \cdots, P$;

Q_i: the ith segmented queue, $i = 1, 2, \cdots, S$;

$FS(q)$: frontend segment[3] of a segmented queue q, where q denotes any segmented queue. For example, in Figure 4 (c), $FS(Q_2)$ = {f, P6} ;

$NS(q)$: number of segments in the segmented queue q;

$L(q)$: length of queue q, i.e. the total number of nodes in queue q;

$M(g)$: number of nodes mapped onto pipeline stage g;

N_r: number of remaining nodes to be mapped onto pipeline stages;

P_r: number of remaining pipeline stages for nodes to be mapped onto.

First, we convert each subtrie to a segmented queue. For each subtrie ST_i, we allocate a queue Q_i, $i = 1, 2, \cdots, S$. Then we do a Breadth-First Traversal of each subtrie. Except the root of the subtrie, each traversed node is pushed into the corresponding queue. In between any two nodes on different levels, a border mark is set on the queue. Hence each queue is partitioned into several segments. The data structure of each segment is a queue as well. The nodes on the same level of the subtrie belong to the same segment. At this time, we have S segmented queues with (possibly different) lengths $L(Q_i)$, $i = 1, 2, \cdots, S$.

Input: a leaf-pushed uni-bit subtrie with root R
Output: a segmented queue Q that has sn segments and qn nodes
initialize a node pointer p;
$p \leftarrow R$;
$sn \leftarrow 1, qn \leftarrow 1$;
push p into the sn th segment;
while *the sn th segment has nodes* **do**
  $sn \leftarrow sn + 1$;
  while *any node in the $(sn-1)$ th segment has not been retrieved* **do**
    p $\leftarrow$ node retrieved from the $(sn-1)$ th segment;
    if *p has children* **then**
      push p's left-child into the sn th segment;
      push p's right-child into the sn th segment;
      $qn \leftarrow qn + 2$;
    end
  end
  link the sn th segment to the end of $(sn-1)$ th segment;
end
Q includes the $2nd, 3rd, \cdots, (sn-1)$ th segments;
return Q;

Algorithm 2: Converting a subtrie to a segmented queue.

[3] A segment is actually a queue.

Next we pop the nodes from the queues and fill them into the memory of the pipeline stages. The pipeline is filled from the second stage since the first stage is dedicated to the roots of the subtries. Hence the initial value of P_r is $P-1$. There are $N_r = \sum_{i=1}^{S} L(Q_i)$ nodes in all to be filled into the pipeline. A balanced pipeline should have $\lceil N_r/P_r \rceil$ nodes in each stage except possibly the first stage. We sort the S queues in decreasing order of the number of segments in each queue and the number of nodes in their frontend segments, so that (1) $NS(Q_i) \geq NS(Q_j)$, and (2) $L(FS(Q_i)) \geq L(FS(Q_j))$ if $NS(Q_i) = NS(Q_j)$, $1 \leq i < j \leq S$. Then we pop nodes from queues in order. Only the frontend segments are allowed to pop nodes. For any queue, if all the nodes in the frontend segment have been popped, then the queue is not allowed to pop any more nodes until we move on to the next stage. We pop nodes until either there are $\lceil N_r/P_r \rceil$ nodes in this stage, or no more frontend segments can be popped. For example, in Figure 4 (c), when filling Stage 2, we pop nodes out of Q_2 first. After popping node f and P6, no more node can be popped out of Q_2, since all the nodes in its frontend segment are popped out. After P2 is popped from Q_1, Stage 2 is full and we move on to Stage 3. When filling Stage 3, all frontend segments are updated, allowing Q_2 to pop nodes again.

Input: S subtries: $ST_1, ST_2, \cdots, ST_S$
Input: P pipeline stages: $SG_1, SG_2, \cdots, SG_P$
Output: filled pipeline
create and initialize S queues;
$N_r \leftarrow 0, P_r \leftarrow P-1$;
for $i \leftarrow 1$ *to* S **do**
  convert ST_i to Q_i ;
  $N_r \leftarrow N_r + L(Q_i)$;
end
for $i \leftarrow 1$ *to* P **do**
  $M_i \leftarrow N_r/P_r$;
  update $FS(Q_j)$, $j = 1, 2, \cdots, S$;
  sort $Q_1, Q_2, \cdots, Q_S$ in the decreasing order of $NS(Q_j)$ and $L(FS(Q_j))$, $j = 1, 2, \cdots, S$;
  while $M(SG_i) < M_i$ **do**
    pop nodes from queues in order and fill into SG_i;
    if *no queue can pop a node* **then**
      break;
    end
  end
  $N_r \leftarrow N_r - M(SG_i), P_r \leftarrow P_r - 1$;
end

Algorithm 3: Mapping S subtries onto P pipeline stages.

Figure 4 (c) illustrates the outcome of the mapping procedure.

3.4 Skip-enabling in the Pipeline

To support the above flexible mapping, we need to implement the *nop* (no-operation) in the pipeline. Our method is simple. 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 for prefix 111 in Stage 1, we retrieve (1) the node P8's memory address in Stage 4, and (2) the distance from Stage 1 to Stage 4. When a request is passed through the pipeline, the distance value is decremented. When the distance value becomes 0, the child node's address is used to access the memory in that stage.

3.5 Analysis and Comparison

In the proposed architecture, a lookup can be performed in each clock cycle. The delay for each lookup is constant, measured as the number of clock cycles, equal to the number of pipeline stages. The memory requirement is proportional to the total number of subtrie nodes. Since OLP has the same entry point and unique exit point, it keeps the output sequence in the same order as input. As a linear pipeline architecture, OLP can use the same update scheme as proposed in [2]. By inserting *write bubble*, the pipeline memory can be updated without disrupting the on-going operations.

We use the following notations for performance analysis.

P: number of pipeline stages;

N: total number of trie nodes;

L: request queue size in CAMP.

Based on the discussion in the previous sections, we compare the performance of the OLP with CAMP and the Ring pipeline in Table 2. The metrics are defined as follows.

- **Throughput**. It is defined as the number of lookups per clock (LPC).
- **Memory**. It includes the memory used by all the pipeline stages and additional memory if any, such as the request queues in CAMP.
- **Delay**. It is the number of clock cycles to go through the entire IP lookup engine.
- **Keep in order**. It means that the order of the output packets is same as the input order.
- **Disruption-free update**. It means the pipeline operations won't be disrupted by route updates.

Table 2 shows performance comparison between OLP, CAMP and the Ring pipeline.

Table 2. Performance Comparison

Solution	OLP	CAMP	Ring
Throughput (LPC)	1	0.8	0.5
Total memory	$O(N)$	$O(N+P)$	$O(N)$
Worst-case delay	P	$O(PL)$	$2P$
Keep in order	Yes	No	Yes
Disruption-free update	Yes	No	No

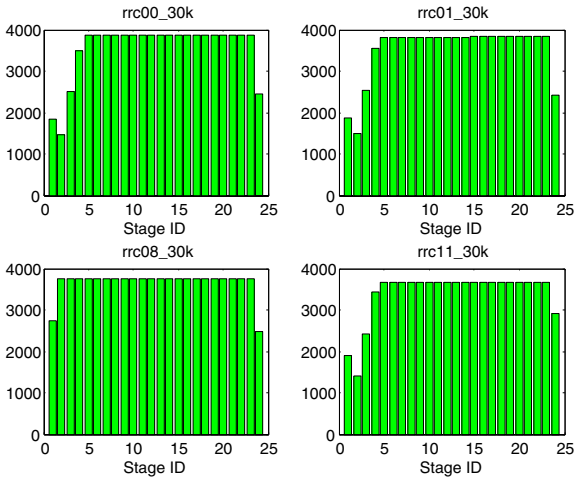

Figure 5. Trie node distribution over the pipeline stages in the OLP mapping.

4 Experimental Results and FPGA Implementation

4.1 Experimental Results

To verify the OLP mapping scheme, we conducted experiments on prefix sets from four routing tables rrc00, rrc01, rrc08 and rrc11 [12]. Each prefix set contained $30K$ prefixes. We parsed each set and obtained its corresponding leaf-pushed uni-bit trie which has over 80K nodes. We used the OLP scheme to map trie nodes to a 24-stage pipeline and observe the trie node distribution over the pipeline stages. Figure 5 shows the experimental results for the four prefix sets. Each pipeline stage stores less than 4K nodes.

4.2 Implementation and Comparison

We have implemented OLP on a Xilinx Virtex II Pro FPGA device XC2VP70 [19]. It has 5.9 Mb of BlockRAM and its maximum operating frequency is 350 MHz. The abstracted schematic of OLP is shown in Figure 6. There are

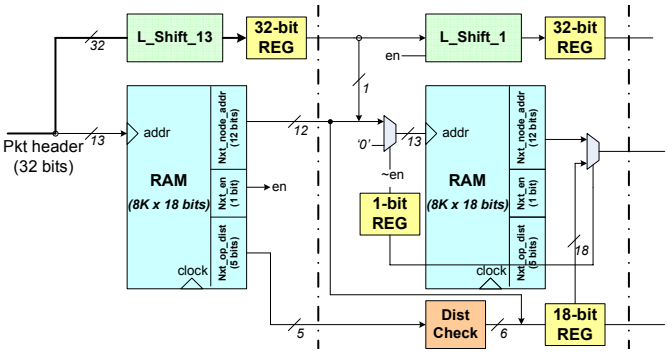

Figure 6. OLP schematic for FPGA implementation.

Table 3. Experimental Results for OLP, CAMP and Ring

Solution	OLP	CAMP	Ring
Throughput (Gbps)	80	64	40
Memory (Mb)	3.456	3.888	3.492
Delay (ns)	96	80 $\sim$ 376	192
Slices used	362	528	408

24 pipeline stages. Each stage has a RAM of $8K \times 18$ bits, i.e. 8 BlockRAMs, sufficient to store 4K trie nodes. Based on the results in Section 4.1, such a FPGA device can contain a routing table of 30K prefixes.

We have also implemented CAMP and the Ring pipeline architectures on the same FPGA device. Due to space limitation, we do not present the details here. The abstracted schematics are shown in Figures 7 and 8.

OLP, CAMP and the Ring pipeline are implemented in Verilog. The code was synthesized using Xilinx ISE 8.2i XST. Post place and route results are shown in Table 3. All the three architectures achieve a clock rate of 250 MHz on the above specified FPGA device.

OLP achieves throughput of 250 million lookups per second, that is, 80 Gbps for the minimum packet size of 40 bytes. OLP has twice the throughput of the Ring pipeline. Compared with CAMP, in the worst case, when LPC is 0.8, OLP achieves 25% higher throughput. In addition, OLP has the smallest logic area and the shortest delay.

5 Conclusion

Pipelined architecture for trie-based IP lookup is a promising solution for high speed routers. This paper pro-

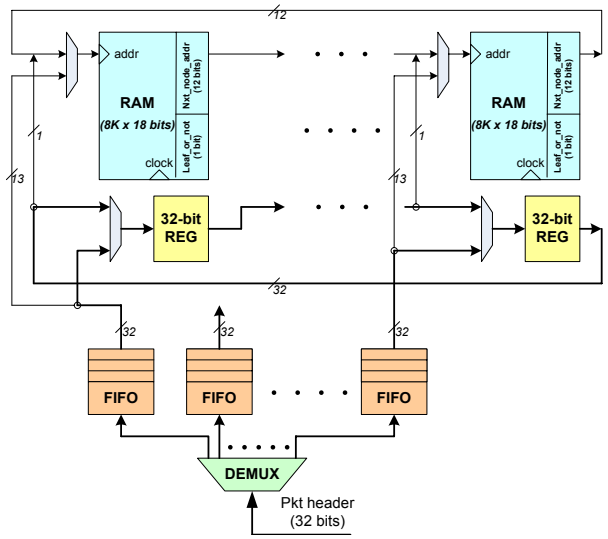

Figure 7. CAMP schematic for FPGA implementation.

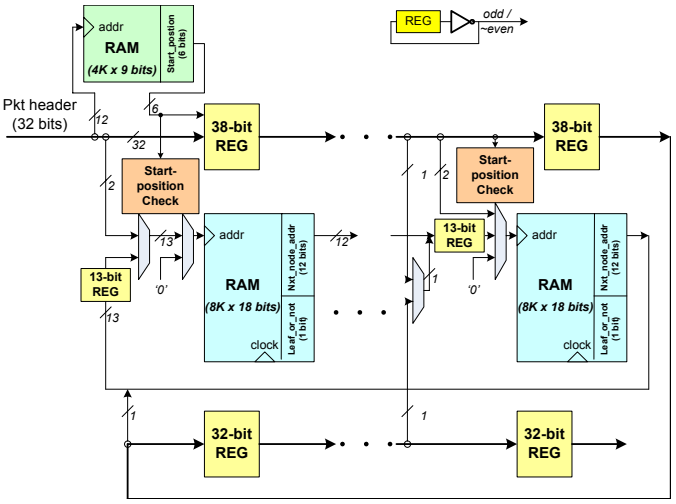

Figure 8. Ring schematic for FPGA implementation.

posed a novel linear pipeline architecture for trie-based IP lookup. It offers more freedom in mapping trie nodes to pipeline stages by supporting *nop*s. Any two nodes on the same level of a sub-trie can be mapped to different pipeline stages. Thus it can achieve an evenly distributed memory while maintaining high throughput of one lookup per clock cycle. Also, it has constant delay and maintains input order. We implemented our design as well as two state-of-the-art designs on commodity FPGAs. Compared to the two reference solutions, our design improves the throughput by 25% and 100%.

References

[1] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. *Proceedings of ISCA '05*, pages 123–133, Jun 2005.

[2] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. *Proceedings of INFOCOM '03*, 1:64–74, Mar/Apr 2003.

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

[4] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *Proceedings of SIGCOMM '97*, pages 3–14, Sep 1997.

[5] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. *IEEE/ACM Transactions on Networking*, 14(2):397–409, Apr 2006.

[6] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Computer Communication Review*, 34(2):97–122, Apr 2004.

[7] H. Fadishei, M. S. Zamani, and M. Sabaei. A novel reconfigurable hardware architecture for IP address lookup. *Proceedings of ANCS '05*, pages 81–90, Oct 2005.

[8] K. S. Kim and S. Sahni. Efficient construction of pipelined multibit-trie router-tables. *IEEE Transactions on Computers*, 56(1):32–43, Jan 2007.

[9] S. Kumar, M. Becchi, P. Crowley, and J. Turner. Camp: fast and efficient IP lookup architecture. *Proceedings of ANCS '06*, pages 51–60, Dec 2006.

[10] W. Lu and S. Sahni. Packet forwarding using pipelined multibit tries. *Proceedings of ISCC '06*, pages 802–807, Jun 2006.

[11] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, Jun 1999.

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

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

[14] S. Sahni and K. S. Kim. Efficient construction of multibit tries for IP lookup. *IEEE/ACM Transactions on Networking*, 11(4):650–662, Aug 2003.

[15] S. Sikka and G. Varghese. Memory-efficient state lookups with fast updates. *SIGCOMM Computer Communication Review*, 30(4):335–347, Oct 2000.

[16] H. Song, J. Turner, and J. Lockwood. Shape shifting trie for faster IP router lookup. *Proceedings of ICNP '05*, pages 358–367, Nov 2005.

[17] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 17(1):1–40, Feb 1999.

[18] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. *Proceedings of SIGCOMM '97*, pages 25–38, Sep 1997.

[19] Xilinx Virtex-II Pro FPGAs. http://www.xilinx.com.