This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE INFOCOM 2010 proceedings

Towards Practical Architectures for SRAM-based Pipelined Lookup Engines

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

***Abstract*—Lookup engines for various network protocols can be implemented as tree-like search structures. Mapping such search structures onto static random addressable memory (SRAM) -based pipeline architectures has been studied as a promising alternative to ternary content addressable memory (TCAM) for high performance lookup engines in next generation routers. Incremental update and memory balancing are identified as two of the major challenges for the SRAM-based pipelined solutions to be practical. Although these two challenges have been *separately* addressed in some previous work, whether they can be solved *simultaneously* remains a question. Most of the existing mapping schemes to achieve balanced memory distribution across pipeline stages are *static*, where the entire search structure needs to be re-mapped to the pipeline once the lookup table has been updated. This paper takes IP lookup as a case study and proposes the *incremental mapping* scheme to support incremental updates while preserving balanced memory distribution across stages in a linear pipeline. We discuss two variants of the scheme and evaluate their performance in terms of memory balancing and update cost. Furthermore, we optimize the scheme to enable interfacing with external SRAMs so that even larger routing table can be supported using a single chip with limited on-chip memory. Simulation using real-life routing tables validates our schemes. Prototype on a state-of-the-art field programmable gate array (FPGA) shows that our architecture can sustain 80 Gbps throughput for minimum size packets while supporting the currently largest routing table. We believe the proposed schemes can be applied to other SRAM-based pipelined lookup engines for various network protocols.**

I. Introduction

The kernel function of network routers/switches is to forward packets based on the results of some form of lookup. While various network protocols are deployed, most of them can be implemented as tree-like search structure which can be pipelined to achieve high throughput. Explosive growth of network traffic results in packet lookup as a long-time performance bottleneck for network routers [1], [2]. Hardware-based solutions such as ternary content addressable memory (TCAM) [3] thus become a necessity to meet the throughput requirement. Due to their flexibility, high throughput and low power consumption, static random addressable memory (SRAM) -based pipeline architectures have been developed as a promising alternative to TCAMs for high performance lookup engines in next generation routers [4]–[6]. Supporting incremental update and achieving balanced memory utilization across the stages have been identified as two of the major challenges for SRAM-based pipelined solutions to be practical [4]. Over the past few years, various schemes have been proposed to tackle these two challenges *separately* [4], [5], [7]–[9]. However, one question remains unanswered: How can a pipelined lookup engine support dynamic incremental update while preserving balanced memory distribution across the stages?

Since most of the lookup problems share the similarity with IP lookup [6], we take SRAM-based pipelined IP lookup engines as a case study to address the above challenges. Pipelined IP lookup engines are usually based on various routing tries [10]. Most of the existing trie-to-pipeline mapping schemes to achieve balanced memory distribution are *static*: All the trie nodes have to be re-distributed to the pipeline stages once the trie has any change. In other words, they do not support incremental update.

In addition, one question needs rethinking: Is it always desirable to balance the memory distribution across the pipeline stages? Off-chip memories have to be used to support large routing tables in real-life ASIC design. With limited I/O pins, it is not practical to employ a large number of external memories. A flexible trie-to-pipeline mapping scheme that controls (rather than balances) the memory distribution across the pipeline stages becomes more desirable.

In this paper we present our efforts to achieve a practical SRAM-based pipeline architecture for high performance IP lookup. First, we propose an incremental mapping scheme to simultaneously address the two challenges: incremental update and memory balancing. Two variants of the scheme are also discussed. Second, we extend our mapping scheme to support larger routing tables by using few external memories. Third, we validate our solution via simulation and the prototype on a field programmable gate array (FPGA). The results show that our solution achieves controllable memory distribution across the pipeline stages while supporting dynamic incremental route update. The FPGA implementation for a backbone routing table sustains 80 Gbps throughput for minimum size (40 bytes) packets.

The rest of the paper is organized as follows. Section II gives a brief overview of routing tries and SRAM-based pipelined IP

This work is supported by the United States National Science Foundation under grant No. CCF-0702784. Equipment grant from Xilinx Inc. is gratefully acknowledged.

978-1-4244-6739-6/10/$26.00 ©2010 IEEE

lookup engines. Section III discusses our incremental mapping scheme and its variants. Section IV evaluates the performance of our solution and shows the FPGA prototype results. Section V concludes the paper with remarks on the future work.

II. Background and Related Work

A. Trie-based IP Lookup

The entries in the routing table are specified using prefixes, and IP lookup is to find the longest matching prefix for an input IP address. The most widely used data structure in algorithmic solutions for IP lookup is some form of trie [10]. A trie is a binary tree, where a prefix is represented by a node. The value of the prefix corresponds to the path from the root of the tree to the node representing the prefix. The branching decisions are made based on the consecutive bits in the prefix. A trie is called a uni-bit trie if only one bit at a time is used to make branching decisions. Figure 1 (b) shows the uni-bit trie for the prefix entries in Figure 1 (a). Each trie node contains both the represented prefix and the pointer to the child nodes.

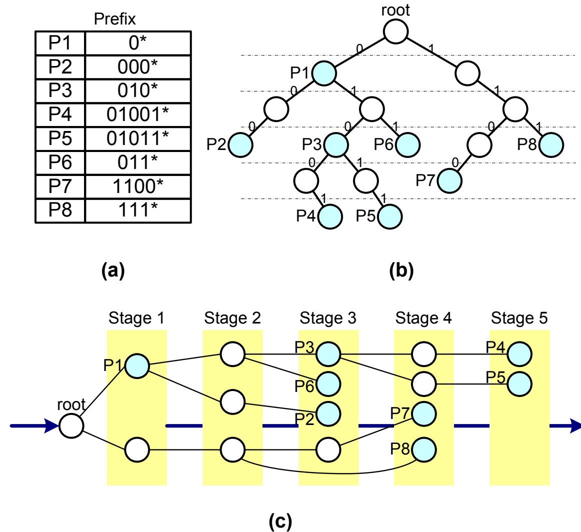

Fig. 1. (a) Prefix entries; (b) Uni-bit trie; (c) Fine-grained trie-to-pipeline mapping.

Given a uni-bit trie, IP lookup is performed by traversing the trie according to the bits in the IP address. The information about the longest prefix matched so far, is carried along the traversal. Thus, when a leaf is reached, the longest matched prefix along the traversed path is returned. The time to look up a uni-bit trie is equal to the prefix length. A multi-bit trie uses multiple bits in one scan to increase the search speed. The number of bits scanned at a time is called the *stride*. Multi-bit tries using a larger stride usually result in much larger memory requirement, while some optimization schemes have been proposed for memory compression [1], [11]. In this work-in-progress paper, we discuss only the uni-bit trie for IPv4, leaving other advanced trie structures as the next step.

B. SRAM-based Pipelined Engines

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 separate stage, so that a lookup request can be issued every clock cycle [4], [12]. However, such a simple scheme results in unbalanced memory distribution across the pipeline stages. This has been identified as a dominant issue for SRAM-based pipeline architectures [7]. In an unbalanced pipeline, the global clock rate is determined by the access time to the "fattest" stage. Since it is unclear at hardware design time which stage will be the "fattest", we must allocate memory with the maximum size for each stage. Such an over-provisioning results in memory wastage and excessive power consumption [7]. Although various pipeline architectures [5], [7], [8] have been proposed recently, most of them balance the memory distribution across stages at the cost of lowering the throughput. Our previous work [5] proposes a fine-grained node-to-stage mapping scheme for linear pipeline architectures. It is based on the heuristic that allows any two nodes on the same level of a trie to be mapped onto different stages of a pipeline. Balanced memory distribution across pipeline stages is achieved, while a high throughput of one packet per clock cycle is sustained. Figure 1 (c) shows the mapping result for the uni-bit trie in Figure 1 (b) using the fine-grained mapping scheme. This paper adopts the same heuristic for mapping a trie onto the pipeline architecture.

III. Practical Pipeline Architectures

Route update (i.e. adding, changing or removing route entries in a routing table) in most of the existing pipelined IP lookup engines involves two phases: (1) trie update and (2) re-mapping the *entire* trie onto the pipeline. Each single update will trigger the re-mapping of the entire trie. This results in high update cost. We focus on route insertion, as route change or removal can be implemented by finding and re-labeling the entries [13]. Our key idea is to map onto the pipeline only the trie nodes that are newly inserted due to route update.

A. Incremental Mapping

We embed the trie-to-pipeline mapping procedure into the trie update (i.e. route insertion). Such a scheme can even enable the online update in hardware architecture. Figure 2 shows the complete algorithm for inserting a prefix into a uni-bit trie. When a new node is created and to be inserted, we map this node onto the pipeline based on the memory distribution across stages at that time. As shown in Figure 3, the mapping must consider the remaining bits in the prefix. For example, the current bit under consideration is the cth bit of the prefix and there are m bits remaining in the prefix for trie extension. If we only focus on the current bit and map it to the stage with lowest memory utilization, it is possible that the bit is mapped to the last stage so that the remaining m bits will not be able to be mapped onto the pipeline. Hence, we find the m stages with the lowest memory utilization and map the current bit onto the first of the m stages. The complexity of the mapping algorithm is $O(mh)$.

Input: r: root of the trie.
Input: p: an inserted prefix whose length is $len(p)$.
Input: H: pipeline depth i.e. the number of stages in the pipeline.
Output: r: root of the updated trie of which the nodes have been mapped to the pipeline.

```
1:  i ← len(p)
2:  n ← r
3:  while i ≥ 0 do
4:     if n.child[p[i]] == null then
5:        Break.
6:     else
7:        n ← n.child[p[i]]
8:     end if
9:     i − −
10: end while
11: // Extend the trie using remaining bits of p.
12: while i ≥ 0 do
13:    n.child[p[i]] ← new node.
14:    n ← n.child[p[i]]
15:    if Conservative then
16:       n.stage ← Map(n.parent.stage + 1, i, H − (32 − len(p)))
17:    else {Aggressive}
18:       if n.parent.stage + i > H then
19:          reMap(n.parent, 1, H − i)
20:          n.stage ← Map(n.parent.stage + 1, i, H)
21:       end if
22:    end if
23:    i − −
24: end while
```

Fig. 2. Algorithm: Inserting a new prefix.

Input: s: the first stage that the node can be mapped to.
Input: m: the number of the remaining bits of the prefix.
Input: h: the number of stages where the node can be mapped.
Input: $Capacity[]$: the capacity of each stage.
Output: g: the index of the pipeline stage to which the node is mapped.

```
1: //Find the m stages with the maximum capacity
2: Q ← {s, s + 1, · · · , s + h − 1}
3: for i ← 1 to m do
4:    t[i] ← arg max_{j∈Q} Capacity[j]
5:    Q ← Q − t[i]
6: end for
7: g ← min_{i=1}^{m} t[i]
```

Fig. 3. Function: $Map(s, m, h)$.

Another issue we must consider is the correlation between different prefixes. For example, there are two prefixes p_1 and p_2 while p_1 is the prefix of p_2. If p_1 is inserted ahead of p_2, it is possible to map the last bit of p_1 to the last stage. As a result, it is impossible to map the last few bits of p_2 to the pipeline, without remapping the bits of p_1. Hence, we develop

Input: n: the node to be remapped.
Input: t: the index of the last stage to which the node can be mapped.
Input: m: the number of the bits involved.
Output: n is remapped.

```
1: if n.stage == 0 then
2:    Return.
3: end if
4: if n.parent.stage + m > t then
5:    reMap(n.parent.stage, m + 1, t − m)
6: end if
7: n.stage ← Map(n.parent.stage + 1, m, t)
```

Fig. 4. Function: $reMap(n, m, t)$.

two variants of the incremental mapping scheme. First, we consider the prefix length of the current prefix being inserted. If the prefix length is $L < W$ where W is the maximum prefix length (e.g. $W = 32$ for IPv4), we reserve the last $W - L$ stages for other longer prefixes and will not map any bit of the current prefix onto those stages. Such a method is *conservative*. Another method is more *aggressive*: We allow the bits of the current prefix to be mapped to all available stages. When the prefix correlation issue occurs, we remap the bits of the shorter prefix to preceding stages so that the bits of the longer prefix can be mapped onto the pipeline. Figure 4 shows the recursive implementation of the remap algorithm.

It is expected that the aggressive method can achieve more balanced memory distribution across the stages, at the cost of additional time for remapping. However, the number of nodes involved in the remap procedure is bounded by the length of the shorter prefix which is $\leq W$. The complexity of the remapping algorithm is $O(W^3)$.

B. External SRAM

To achieve balanced memory distribution across stages, each stages is assigned equal capacity. However, due to limitation of the available number of I/O pins, it is impractical to let the memory in all stages be equally large for supporting very large routing tables. A practical pipeline architecture should contain few large stages that can be stored in external memory while the remaining stages reside on-chip. The memory utilization among the large stages still needs to be balanced. Hence we extend the fine-grained mapping scheme to control the memory distribution by assigning large or small capacities for corresponding stages.

The question is: Which stage should be assigned to be the large one? Note that the pipeline depth H must be $> W - I$ where I is the number of initial bits for prefix expansion [5]. The number of nodes in stage k ($k = 1, 2, \cdots, H$) is no more than the number of prefixes whose length is longer than $k + I$. We examine the prefix length distribution of the routing tables and find that the majority of the prefixes have a length around 24 (Figure 5). Hence, if we use $I = 8$ for prefix expansion, the few stages around stage 16 tend to contain much more nodes than other stages. We can assign these few stages to be the external stages.

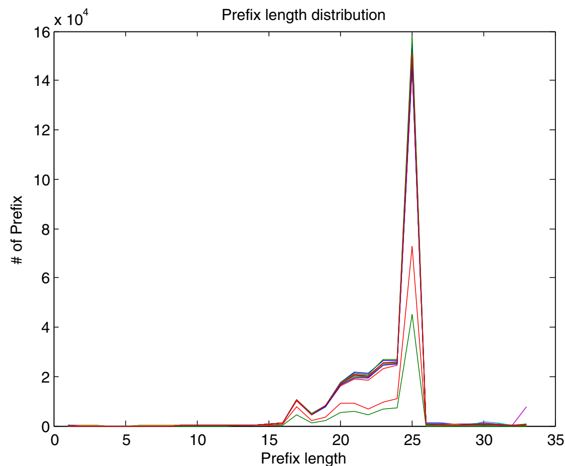

Fig. 5. Prefix length distribution of the 16 routing tables given in Table I.

IV. Performance Evaluation

To validate the effectiveness of our solution, we conducted both software simulation and hardware prototyping using a set of real-life backbone routing tables.

A. Simulation Results

We used 17 real-life backbone routing tables from the Routing Information Service (RIS) [14]. Their characteristics is shown in Table I. Note that the routing tables rrc08 and rrc09 are much smaller than others, since the collection of these two data sets ended on September 2004 and February 2004, respectively [14].

TABLE I
Representative Routing Tables (snapshot on 2009/04/01)

Routing table	# of prefixes	# of prefixes w/ length < 16
rrc00	300365	2366 (0.79%)
rrc01	282852	2349 (0.83%)
rrc02	272504	2135 (0.78%)
rrc03	285149	2354 (0.83%)
rrc04	294231	2381 (0.81%)
rrc05	284283	2379 (0.84%)
rrc06	283835	2337 (0.82%)
rrc07	280786	2347 (0.84%)
rrc08	83556	495 (0.59%)
rrc09	132786	991 (0.75%)
rrc10	283573	2347 (0.83%)
rrc11	282761	2350 (0.83%)
rrc12	284469	2350 (0.83%)
rrc13	289849	2355 (0.81%)
rrc14	278750	2302 (0.83%)
rrc15	299211	2372 (0.79%)
rrc16	288218	2356 (0.82%)

First, we evaluated the performance of the incremental mapping scheme, with respect to its results on memory balancing and update cost for different routing tables. To mimic the route update procedure for each routing table, we inserted the prefixes, one by one, into the trie. Figure 6 shows the final snapshots of the memory distribution across stages for different routing tables. The aggressive incremental mapping achieved almost the same memory balancing as the static mapping, while the conservative incremental mapping was not efficient in utilizing the last few stages due to the conservative reservation. It is hard to balance the first few stages since there are few nodes at the top levels of tries. But since the maximum numbers of nodes at the top levels are small, we can assign the maximum values for their capacities, i.e. 2^i nodes for the ith stage ($i = 0, 1, \cdots, 8$), without much memory wastage.

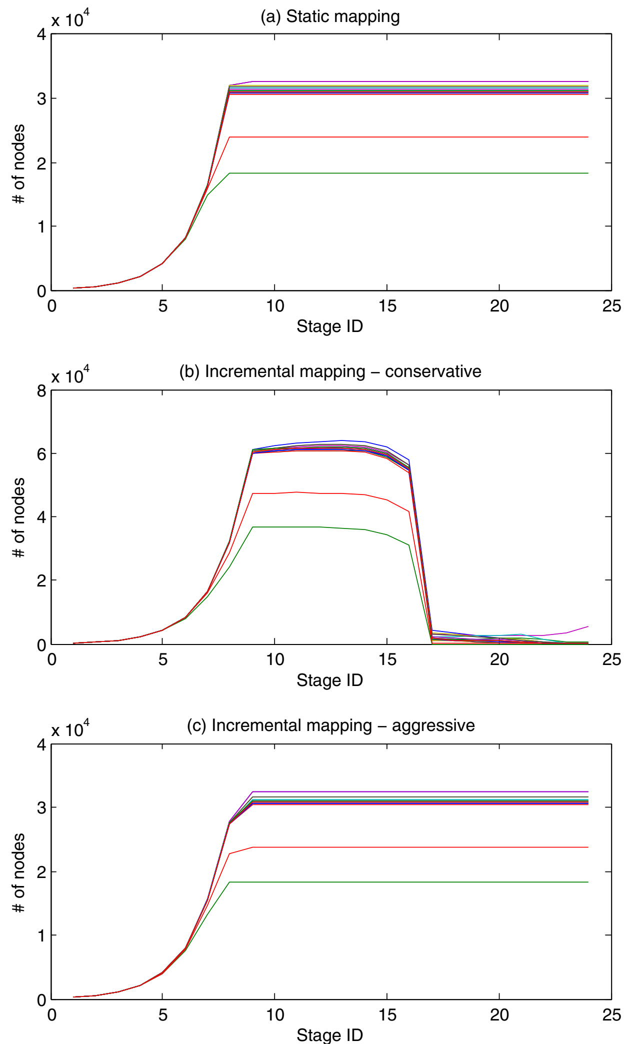

Fig. 6. Mapping results for the 17 routing tables. (a) Static (b) Incremental-Conservative (c) Incremental-Aggressive

We measured the update cost in terms of the computation time to complete the mapping of the entire routing table onto the pipeline. The results shown in Table II were obtained on a desktop with Dual-Core 2.8 GHz CPU and 2GB RAM. It is clear that the incremental mapping outperformed the static mapping with respect to the update cost. In most cases, the update time of the incremental mapping was less than 0.1% of that of the static mapping. Here we do not show the update time of the conservative incremental mapping since it was very close to that of the aggressive version.

To fit the current largest routing table in real hardware, we added 6 external stages to the 24-stage pipeline architecture.

TABLE II
TIME (SECONDS) CONSUMED FOR MAPPING THE ROUTING TABLE ONTO A 24-STAGE PIPELINE

Routing table	Static mapping	Incremental mapping (Aggressive)
rrc00	3090	2
rrc01	2688	2
rrc02	2634	2
rrc03	2837	< 1
rrc04	2920	2
rrc05	2696	2
rrc06	2772	1
rrc07	2770	1
rrc08	942	1
rrc09	1613	2
rrc10	2786	< 1
rrc11	2806	2
rrc12	2947	3
rrc13	2949	2
rrc14	2714	1
rrc15	2782	2
rrc16	2739	1

As discussed in Section III-B, stages 13 $\sim$ 18 were assigned to be stored in off-chip memory. Thus we obtained the memory distribution across the stages, as shown in Figure 7. The memory utilization among the large stages as well as among the remaining stages was balanced.

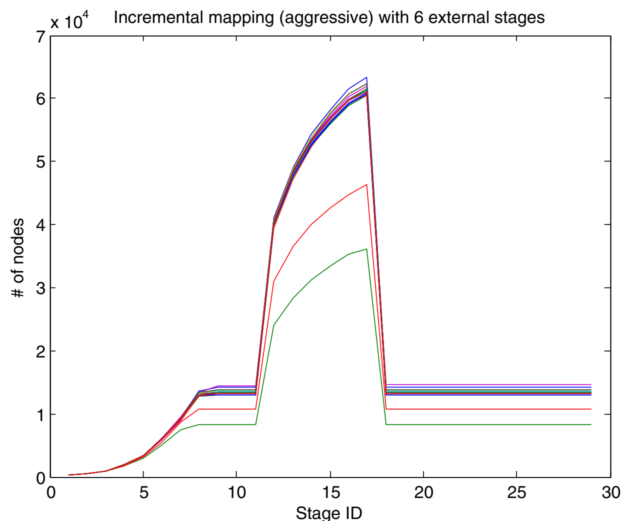

Fig. 7. Incremental mapping results with 6 external stages

B. FPGA Prototype

We prototyped our design with 6 external stages that supported the largest routing table (rrc00) on FPGA using Xilinx ISE 10.1 development tools. The target device was Xilinx Virtex-5 XC5VSX240T with -2 speed grade. Table III shows the post place and route results. The designs achieved a clock frequency of 125 MHz. Dual-port RAMs were used so that two packets could be processed per clock cycle. Assuming the external SRAM can sustain 125 MHz clock frequency (which is available in current market [15]), this results in a throughput of 80 Gbps for minimum size (40 bytes) packets, which doubles the current backbone network rate.

TABLE III
RESOURCE UTILIZATION

	Used	Available	Utilization
Number of Slices	769	37,440	2%
Number of bonded IOBs	316	960	33%
Number of Block RAMs	431	516	83%

V. CONCLUDING REMARKS

This paper presents our efforts in addressing the two main challenges for pipelined lookup engines to be practical: incremental update and memory balancing. We propose an incremental mapping scheme to tackle the two challenges simultaneously. The key idea is to embed the mapping procedure into the trie update. We also extend the mapping scheme to enable using external memory to support larger routing table in real hardware. Simulation experiments and FPGA prototype demonstrate the effectiveness of our solution.

There remain various open issues for SRAM-based pipelined lookup engines. As IPv4 runs out, we need to take into account IPv6 as the basis of the next generation Internet routing. We are also applying our ideas to other lookup tasks in packet processing that can be implemented in SRAM-based pipelines [6]. Supporting in-architecture updates with little help from software [16] is also a promising topic to explore.

REFERENCES

[1] 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.
[2] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman, “Scalable IP lookups using shape graph,” in *Proc. ICNP*, 2009.
[3] 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.
[4] A. Basu and G. Narlikar, “Fast incremental updates for pipelined forwarding engines,” *IEEE/ACM Trans. Netw.*, vol. 13, no. 3, pp. 690–703, 2005.
[5] W. Jiang, Q. Wang, and V. K. Prasanna, “Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup,” in *Proc. INFOCOM*, 2008, pp. 1786–1794.
[6] L. D. Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam, “Plug: flexible lookup modules for rapid deployment of new protocols in high-speed routers,” in *Proc. SIGCOMM*, 2009.
[7] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, “A tree based router search engine architecture with single port memories,” in *Proc. ISCA*, 2005, pp. 123–133.
[8] S. Kumar, M. Becchi, P. Crowley, and J. Turner, “CAMP: fast and efficient IP lookup architecture,” in *Proc. ANCS*, 2006, pp. 51–60.
[9] K. S. Kim and S. Sahni, “Efficient construction of pipelined multibit-trie router-tables,” *IEEE Trans. Comput.*, vol. 56, no. 1, pp. 32–43, 2007.
[10] 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.
[11] H. Song, J. Turner, and J. Lockwood, “Shape shifting trie for faster IP router lookup,” in *Proc. ICNP*, 2005, pp. 358–367.
[12] J. Hasan and T. N. Vijaykumar, “Dynamic pipelining: making ip-lookup truly scalable,” in *Proc. SIGCOMM*, 2005, pp. 205–216.
[13] H. Le, W. Jiang, and V. K. Prasanna, “Scalable high-throughput SRAM-based architecture for IP-lookup using FPGA,” in *Proc. FPL*, 2008, pp. 137–142.
[14] RIS Raw Data, “http://data.ris.ripe.net.”
[15] SAMSUNG High Speed SRAMs, “http://www.samsung.com.”
[16] F. Kiyak, B. Mochizuki, E. Keller, and M. Caesar, “Better by a HAIR: Hardware-amenable internet routing,” in *Proc. ICNP*, 2009.