A Memory Efficient IPv6 Lookup Engine on FPGA

Da Tong
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Email: datong@usc.edu

Yi-Hua E. Yang
Broadcom Corp.
Santa Clara, CA
Email: edyang@broadcom.com

Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Email: prasanna@usc.edu

Abstract—High-speed IP lookup remains a challenging problem in next generation routers due to the ever increasing line rate and routing table size. The evolution towards IPv6 results in long prefix length, sparse prefix distribution, and potentially very large routing tables. In this paper we propose a memory-efficient IPv6 lookup engine on Field Programmable Gate Array (FPGA). Static data structures are employed to reduce the on-chip memory requirement. We design two novel techniques: implicit match identification and implicit match relay, to enhance the overall memory efficiency. Our experimental results show that the proposed techniques reduce memory usage by 30%. Using our architecture, state-of-the-art FPGA devices can support 2 copies of IPv6 routing table containing around 330k routing prefixes. Using dual ported BRAM and external SRAM, 4 pipelines can be implemented on a single device, achieving a throughput of 720 million lookups per second (MLPS).

Index Terms—IP lookup, packet forwarding, longest prefix match, binary search tree, perfect hash function.

I. INTRODUCTION

As the Internet grows rapidly, IP lookup becomes a major bottleneck in high-speed routers. In the recent years, 100 Gbps network links and interfaces have been deployed by major service providers and 400 Gbps is becoming the next major milestone. The adoption of IPv6 imposes challenges on existing IP lookup solutions due to the increased prefix length (64-bit) compared with the classic IPv4 (32-bit). The increasing size (i.e. prefix count) of backbone routing tables further makes the IP lookup problem even more challenging.

In recent years, data structures such as binary trie, binary search tree and hash table have been explored to perform IP lookup [11, 10]. Such algorithmic solutions can be mapped on to platforms such as Field Programmable Gate Arrays (FPGAs) to implement pipelined lookup architectures. Achieving 400 Gbps using a single pipeline requires extremely high clock rate. With 64-byte packet size, the lookup engine needs to run at a clock rate greater than 800 MHz in order to support 400 Gbps throughput. However, implementation on current FPGA and memory technologies typically cannot operate at such high clock rates [2]. Therefore using multiple pipelines becomes a natural choice to improve throughput. With the abundant parallelism available on state-of-the-art FPGAs, multiple parallel pipelines can be implemented on a single chip. However the amount of on-chip memory is limited. Therefore efficient utilization of on-chip resources becomes the key issue in multi-pipeline design.

This paper describes a memory-efficient IPv6 lookup engine on FPGA based on the CLIPS [17] architecture. CLIPS is a pipelined multiphase search technique. Each phase performs a fixed-length exact pattern match to locate a potential prefix match up to one or more pre-defined prefix lengths. The last (tail) phase then performs a simple (7-bit) longest prefix match to find the actual IP lookup result. The search technique used in each phase can be chosen independently. In this paper we extend CLIPS to IPv6. To minimize the on-chip memory requirements, we use perfect hash function or complete binary search trees (BST) for the fixed-length search phases, and the compressed tree bitmap [2] for the tail phase. By storing the hash table in the external memory, we minimize the on-chip memory requirement of the most memory consuming phase.

In the original CLIPS implementation [17], pointers are needed to identify the matching entry and to locate the lookup table in the next phase. To minimize the memory usage, instead of following explicit pointers among various phases, we design two pointer-free matching mechanisms: Implicit Match Identification and Implicit Match Relay, to identify the correct lookup paths in a pointer-free manner. The implicit match identification uses the physical memory address of the data structure at each phase to represent any match in the phase. This eliminates the need to explicitly store a match identifier with every lookup entry. The implicit match relay takes the match result in one phase as part of the search pattern in the next phase. As a result, different search paths can be distinguished within a single data structure without requiring pointers from one phase to the next. We have conducted experiments using 9 IPv6 routing tables. They are the largest IPv6 routing tables available to us. The experimental results show that, for the 9 routing tables, nearly 30% memory savings can be achieved compared to a solution without these two memory compaction techniques.

Due to the reduction in the memory requirement, we are able to store two copies of the data structures in BRAM. This enables us to implement 4 parallel search pipelines on one FPGA chip using dual ported BRAM and SRAM. Running at a moderate frequency of 180 MHz, our lookup engine prototype achieves a throughput of 720 MLPS.

The rest of the paper is organized as follows. Section 2 overviews the existing solutions for IP lookup. Section 3 presents in detail the generic algorithms and data structures for CLIPS. Section 4 presents our optimization and the actual data structure and architectures used on FPGA for IP lookup.
Performance evaluation and experimental results are given in Section 5. Section 6 concludes the paper.

II. PRIOR WORK

There has been an extensive research on LPM (Longest Prefix Match [7]) based IP lookup in the recent years. Several hardware accelerated solutions have been proposed. At a high level, these approaches can be categorized into four groups:

1) Linear pattern search in TCAM \[18, 9, 1\]
2) Binary bit traversal in pipelined tries \[2, 11, 13\]
3) Binary value search in pipelined trees \[10\]
4) Exact match in multiple prefix lengths with Bloom filters \[3, 5\] or hash tables \[2, 8, 14\]

In TCAM-based solutions, each prefix is stored in a word. An incoming IP address is compared in parallel with all the active entries in TCAM in one clock cycle. Although TCAM-based solutions are simple, they are also expensive, power-hungry, and do not scale well to long prefix length or large routing table.

The simplest data structure for IP lookup is the binary trie. In binary trie, the path from the root to a node represents the routing prefix. Given a binary trie, IP lookup starts from the root node and continues traversing the trie using the bits of the input IP address. Multiple memory accesses are required to find the LPM; therefore, pipelining is usually used to improve throughput. In tree-based IP lookup, each routing prefix is either converted to an \(L\)-bit (\(L\) maximum prefix length) value range, or expanded to some pre-defined lengths to compare exactly with part of the input address.

Binary search on prefix length was proposed in \[14\], where \(O(L)\) hash tables are used to match routing prefixes of length \(\leq L\) bits. The hash tables are organized as a binary tree, resulting in \(O(\log L)\) hash table accesses per lookup. On FPGA or ASIC, such a hash-based IP lookup scheme has several disadvantages: (a) a large number of hash tables are required to store routing prefixes of different lengths; (b) expensive table-specific optimization and prefix expansion are required to reduce the number of hash tables \[8\]; (c) additional circuits and memory (e.g., CAM) are required to resolve hash conflicts and overflows.

The CLIPS technique was proposed in \[17\]. It has the advantage that the amount of delay does not depend on the prefix distribution and simple memory balance among relatively few phases. The disadvantage of external pointer access and potentially high number of external memory accesses \[17\] will be the major constraints of performance when the architecture is extended to IPv6.

III. OVERALL ARCHITECTURE

A. Combined Length Infix Pipelined Search

Combined length infix pipelined search (CLIPS) is proposed in \[17\]. CLIPS combines both length and infix searches to solve the longest prefix match (LPM) problem. Binary length search \[14\] is performed in \(O(\log L)\) phases, where \(L\) is the maximum prefix length. Binary infix search is performed at each phase, with infix length \(L/2^i\) at Phase \(i\), \(1 \leq i \leq \log L\).

In this paper we use the CLIPS to solve IP lookup problem for IPv6. Figure 1 shows the sequence of length searches CLIPS performs for IPv6. Each colored square represents an infix bit in the IPv6 prefix. Each of the first 3 phases extracts and matches an appropriate infix from the 64-bit input against a local infix table (LIT). If a match is found in a LIT, then the down-right arrow is followed to reach the next phase; otherwise, the down-left arrow is followed. Note that all infixes used in any particular phase have the same length, a distinct feature of CLIPS different from the previous prefix length-search approaches \[8, 14, 15\]. Such length regularity allows the multiple CLIPS phases to be easily pipelined on FPGA.

As can be seen from Figure 1, there are \(2^{d-1} = 8\) “length-paths” to reach Phase 4. The end of each length-path indicates a unique 7-bit length range where the longest matching routing prefix may be found in the original routing table. Instead of matching even shorter infixes at Phase 4, 5 and 7, the three phases are integrated into a single tail phase (the new Phase 4) to perform LPM in the last 7-bit range.

Note that the same process can also be used directly for IPv4 lookup. When an IPv4 routing address is received, we simply skip Phase 1 and go directly to a special lookup in Phase 2. From there on the 32-bit LPM can be found as usual. Effectively, CLIPS provides seamless and efficient processing for both IPv4 and IPv6 routing.

B. Construction

1) Basic data structure: In CLIPS, conceptually, a local infix table (LIT) entry can be described in Figure 2. Each LIT entry is uniquely identified by the \(\text{PrefixID}, \text{InfixBits}\) pair. An entry must be either a routing prefix itself \((\text{IsRtp} = 1)\), a “marker” for some longer routing prefix \((\text{IsMkr} = 1)\), or both. The entry also maps the \(\text{PrefixID}, \text{InfixBits}\) to a MatchID, which is used as a PrefixID in a later phase if the entry is a marker.

To construct the LITs, first each routing prefix is cut into consecutive infixes. The cutting can be performed by walking down a “length-path” in Figure 1 for the routing prefix. A “cut” is made whenever a down-right arrow in Figure 1 is followed. For example, a 44-bit routing prefix specifying bits \([63:20]\) will be cut twice, first at offset 32 to produce infix \([63:32]\) and then at offset 24 to produce infixes \([31:24]\) and \([23:20]\). For routing prefixes of length < 64 bits, the cutting produces at most one infix of lengths 32, 16 and 8, plus optionally a “tail” infix of length 1–7. Each infix is then assigned to a phase according to the infix length. An infix of length \(2^d - i\) is
assigned to Phase $i$, $1 \leq i \leq 3$. The tail infix, if any, is always assigned to Phase 4.

2) Implicit Match Identification: In this paper we design this new technique to reduce the memory requirement. The basic idea of the Implicit Matching Identification is to use the physical address of the LIT entry as the match identifier of that entry so there is no need to explicitly store the MatchIDs discussed above. Both the (PrefixID, InfixBits) pair and the MatchID uniquely identifies the LIT entry. While the PrefixID and the InfixBits are required for correct LIT lookup, the MatchID can be represented implicitly by the physical addresses of the LIT entry, therefore eliminating the extra memory usage required for this field.

In practice, the MatchID at phases 1 through 3 has comparable (usually longer) length than both the PrefixID and the InfixBits. By representing the MatchID implicitly using the physical address of the LIT entry, up to 30% of memory space can be saved at these phases.

C. Longest Prefix Match

1) Basic search process: In the extended CLIPS, once a routing table is converted to the 4 phases, the longest prefix match (LPM) of any input address can be found by going through the 4 phases as described in Algorithm 1.

Initially, the infix [63:32] is used at Phase 1. At each Phase $i$, $1 \leq i \leq 3$, if a “marker” infix is found in the LIT, then the right-adjacent half-length infix is used at Phase $(i + 1)$ (step 10); otherwise, the left-half of the current infix is used (step 15). A matching LIT entry can represent a matching prefix $R$ (step 6) and/or a forward marker $F$ (step 9). If the LIT entry is a matching prefix, then its (implicit) MatchID is kept as a possible LPM output. If the entry is a forward marker, then the MatchID is used as the PrefixID in the lookup at the following phase(s); otherwise, no lookup is needed at any following phase. At Phase 4, a 7-bit infix is used to traverse a compressed tree bitmap Bando and Chao [2] following the MatchID from Phase 3.

2) Implicit Match Relay: In this paper, in addition to Implicit Match Identification, we propose Implicit Match Relay to further reduce memory requirement. The searching process in Algorithm 1 relies on the matching of the PrefixID and InfixBits fields of any LIT entry to the MatchID and $v[b_l : b_r]$ bits of the input address, respectively (step 4). In practice, the two fields can be combined into a shorter bitvector as long as it can still uniquely represent each LIT entry.

More specifically, suppose Phase 1 outputs an (implicit) MatchID of length 20 bits. Conceptually, the PrefixID at Phase 2 would also have 20 bits; together with the 16-bit InfixBits, each LIT entry at Phase 2 would be identified (matched) by 36 bits. However, even for a routing table with several million prefixes, the Phase 2 should have much fewer than $2^{36}$ LIT entries, since each routing prefix produces at most one LIT entry per phase (see Section III-B). Thus it is very likely that much fewer than 20 bits of the MatchID from Phase 1 need to be “relayed” to the PrefixID at Phase 2 to uniquely identify an LIT entry. By combining both PrefixID and InfixBits in a single data structure for this “implicit match relay”, additional space can be saved from the unneeded MatchID bits.

IV. DESIGN OF CLIPS PIPELINE ON FPGA

A. Design Overview

Figure 3 gives an overview of the CLIPS pipeline on FPGA. The infix search is carried out by perfect hash function and pipelined binary search trees (BST) in Phases 1–3 and compressed tree bitmap (CTBM) in Phase 4.
Algorithm 1 64-bit LPM with CLIPS.

Input: 64-bit input bitvector, \textbf{v}[63:0].

Input: LIT(i), 1 \leq i \leq 3, plus a tail search trie.

Output: A longest prefix match, or null if none.

1: Initialize:
   i \leftarrow 1 \text{ (phase index)};
   b_l \leftarrow 63, b_r \leftarrow 32 \text{ (infix boundaries)};
   F \leftarrow 0 \text{ (forward marker)};
   R \leftarrow \text{null} \text{ (prefix match)}.
2: repeat
3:    Set \( l = b_l - b_r + 1 \) \text{ [infix length]}
4:    if \exists \text{ entry } E \in \text{LIT}(i) \text{ such that } E[\text{PrefixID}] = F \text{ and } E[\text{InfixBits}] = v[b_l:b_r] \text{ then}
5:       if \( E[\text{IsRtp}] = 1 \) then
6:          Set \( R \leftarrow E[\text{MatchID}] \).
7:       end if
8:    if \( E[\text{IsMkr}] = 1 \) then
9:       Set \( F \leftarrow E[\text{MatchID}] \).
10:      Set \( b_l \leftarrow b_l - 1, b_r \leftarrow b_r - l/2. \)
11:   else
12:      Set \( F \leftarrow \text{null} \).
13:   end if
14: else \{no matching entry found in LIT(i)\}
15:   Set \( b_r \leftarrow b_r + l/2. \)
16: end if
17: until \( i > 3 \) or \( F = \text{null} \)
18: if \( F \neq \text{null} \) then
19:   Set \( b_l \leftarrow b_l - 1, b_r \leftarrow b_r - 7. \)
20:   Traverse a search trie for \( F \) with \( v[b_l:b_r] \).
21: if a valid leaf node \( N \) is reached then
22:   Set \( R \leftarrow N[\text{MatchID}] \)
23: end if
24: end if
25: end if
26: Use \( R \) as the LPM output.

\( B. \) Phase 1 Perfect Hash Function

The fixed-length exact pattern match in Phase 1 is implemented using perfect hashing. All the LIT entries in Phase 1 are stored in the hash table. The hash value of InfixBits of each LIT entry is used as its index in the hash table, i.e., physical address in the memory. Each entry of the hash table consists of 32 bits InfixBits, 1 bit IsMkr and 1 bit IsRtp. During the search, the perfect hash function will compute the hash value of infix [63:32] of the input IP address. This value is used as the index to access the perfect hash table. If the input bitvector (infix [63:32] of the input IP address) matches the InfixBits in the table entry, then the physical address of that entry will be forwarded to the next phase as MatchID. In our experiments, the maximum MatchID length is 20 bits.

We employ the hash function proposed in [6]. In the construction of the hash table, all LIT entries are first conceptually stored in a two dimensional array of \( 2^{16} \times 2^{16} \). The LIT entry with address \( v \) (which will be smaller than \( 2^{32} \)) is assigned to row \( v/2^{16} \) and column \( v \bmod 2^{16} \). Therefore each possible IP address can find a unique position in this two dimensional array.

To form a one dimensional array, all rows are then "collapsed" onto a linear entry table where each row is given a non-negative "offset" value so that after the offset, no two values in different rows are assigned to the same column. These offset values can be found using exhaustive search. Let the largest row offset be \( F \). \( F \) is bounded by \( 2^{16} \times (2^{16} - 1) \) because the worst case offset would appear when all the rows are concatenated one after another. Each row has 2\( 16 \) entries and there are 2\( 16 \) rows thus the last row will have the worst case offset. In practice the offset will be much smaller than the worst case. For all the routing tables used in our experiment \( F \) is smaller than \( 2^{20} \). The size of the hash table is \( (F + 2^{16}) \times (32 + 2) \) (32-bit InfixBits plus 2 bits IsMkr and IsRtp). This is stored in off-chip SRAM.

Each offset value requires \( \log_2(F) \) bits. There are \( 2^{16} \) rows. Thus BRAM usage is \( 2^{16} \times \log_2(F) \). Since \( F \) is bounded by \( 2^{16} \times (2^{16} - 1), \log_2(F) \) is bounded by 32. Thus the size of the offset table is bounded by \( 2^{16} \times 32 = 2.09 \) Mbits. From the analysis above we can see that the on-chip memory usage does not depend on the size or the prefix distribution of the routing table. By storing the offset values in BRAM and hash
table in off-chip SRAM, we can have a small BRAM usage for any routing table. Note that the actual BRAM usage also depends on the memory word length. On the target device the memory word length must be multiple of 18 bits, thus both 20 and 32 bits offset will occupy 36 bits of BRAM and the offset table will use \(2^{16} \times 36 = 2.36\) Mbits of BRAM.

### C. Phase 2–3: Pipelined BST

We design Phases 2 and 3 as pipelined BST. Each BST stores all the LIT entries of the corresponding phase. An LIT entry in Phase 2 consists of 20 bits PrefixID, 16 bits InfixBits, 1 bit IsMkr and 1 bit IsRtp. Again, 20 bits of implicit MatchID is relayed from Phase 2 to Phase 3. An LIT entry in Phase 3 consists of 20 bits PrefixID (the full MatchID received from Phase 2), 8 bits InfixBits, 1 bit IsMkr and 1 bit IsRtp. Still 20 bits of implicit MatchID is relayed from Phase 3 to Phase 4 (the tail phase).

In the pipelined BST, each pipeline stage stores all the nodes in one level of the BST; each BST node stores one LIT entry. After comparing all the LIT entries with the input bitvector, one of the two possible child nodes in the next BRAM stage will be selected. The child node address is the current node address appended by the 1-bit child selection ("0" or "1"). If a match is found between an LIT entry and the input bitvector, then the physical address of the BST node concatenated by the offset of the LIT entry ("0" or "1") is output as the MatchID to the next phase.

### D. Phase 4: Compressed Tree Bitmap

We implement a 7-bit PC Trie [2] for the multi-bit trie lookup at Phase 4. Every 8 “sibling” nodes in the 7-level trie are represented by a single bit in the bitmap. Thus a ‘1’ in the bitmap represents 8 consecutive prefixes at the same length. The 7-bit trie requires 31 bits for the bitmap [2] (or 32 bits if we ignore the value in the least significant bit position). In addition to the bitmap, a pointer is needed to access the list of next-hop information (NHI). There are 8 NHIs for each ‘1’ in the bitmap. Note that Phase 4 consumes no on-chip BRAM. Both the (bitmap, pointer) and the list of NHIs are stored in external memory.

### V. EXPERIMENTAL RESULTS

#### A. Experimental Setup

Due to the lack of large real-life IPv6 routing tables, we use IPv4 routing tables from Project - RIS [4] captured on 06/03/2010 to generate the corresponding IPv6 routing tables using the same method as proposed in [16]. The IPv4-to-IPv6 prefix mapping is one-to-one. Hence, the number of prefixes in an IPv6 table is the same as its corresponding IPv4 table. The size of the routing tables are shown in Table I.

#### B. Implicit Match Identification and Implicit Match Relay

Implicit match identification and implicit match relay are only applied to the first 3 phases. As shown in Table III, the on-chip memory usage Phase 2 is reduced by 20%. As shown in Table II, the memory requirement of perfect hashing in Phase 1 is reduced by more than 50% due to the application of these two mechanisms. Since applying these two techniques does not affect the on-chip memory requirement of Phase 3 or the external memory requirement of Phase 4, we only show the memory requirements of our implementation using the implicit scheme in the respective tables.

### C. On-chip and Off-chip Memory Usage and Throughput

Our design is implemented on a state-of-the-art FPGA device with 38 Mb on-chip BRAM (Xilinx Virtex 6 XC6VSX475T). External DDR II SRAM is used to store the hash table for Phase 1 and Compressed Tree Bitmap for Phase 4.

The on-chip BRAM used by the design is shown in Table III. For all the 9 IPv6 routing tables (see Table I), using dual-port mode, the same memory block can be shared by two pipelines and thus the available BRAM can support the data structures for 4 pipelines.

Using perfect hashing greatly saves the on-chip BRAM. As shown in Table III, the on-chip BRAM usage of perfect hash function (Phase 1) is much less than the complete BST (Phase 2 and 3) even when the number of LIT entries in Phase 1 is much more than Phase 2 and 3 (The number of LIT entries in Phase 1 is 300–350k, in Phase 2 it is 150–160k, in Phase 3 it is 120–130k).

We have implemented the first 3 phases of our proposed architecture. The Post Place and Route (PAR) results shows that Phase 1 operates at 180 MHz and Phase 2 operates at 299 MHz. Phase 3 has the same circuit architecture as Phase 2 but with less number of pipeline stages. It operates at 300MHz. Phase 4, Compressed Tree Bitmap has been implemented in [2]. According to [2], the compressed tree bitmap can achieve a clock rate of 200 MHz using DDR III SDRAM. Since we use faster DDR II SRAM, the same clock rate can be achieved.

DDR II SRAM can work at 400 MHz with 36-bit data width and burst length of 2 [12]. With the help of dual ported SRAM interface, the SRAM controller allows two read operations per cycle at over 200 MHz. So each SRAM interface can serve 2 CLIPS pipelines to perform one external memory access per cycle. Each CLIPS pipeline requires 3 external memory accesses per cycle (one in Phase 1 and two in Phase 4). Therefore, 4 pipelines can be implemented on the FPGA device using 6 SRAM interfaces. Each SRAM interface requires ~110 pins on the device. In total, the SRAM interfaces of all 4 pipelines use less than 700 I/O pins, well within the number of I/O pins available on a state-of-the-art FPGA device.

Running at 180 MHz, our design can support a lookup rate of 720 MLPS.

### VI. CONCLUSION

In this paper, we design a novel implementation of CLIPS on FPGAs. We use perfect hashing and Complete BST to reduce the on-chip memory consumption and designed two implicit match schemes to reduce the overall memory requirement. As a result 4 pipelines can be implemented on a state-of-the-art FPGA device. The proposed architecture can support
IPv6 routing tables of 330k prefixes, while sustaining a high lookup rate of 720 million lookups per second.

REFERENCES


