Scalable IP Lookup for Programmable Routers

David E. Taylor, John W. Lockwood, Todd Sproull, Jonathan S. Turner, David B. Parlour

Abstract—**Continuing growth in optical link speeds places increasing demands on the performance of Internet routers, while deployment of embedded and distributed network services imposes new demands for flexibility and programmability. IP address lookup has become a significant performance bottleneck for the highest performance routers. New commercial products utilize dedicated Content Addressable Memory (CAM) devices to achieve high lookup speeds. This paper describes an efficient, scalable lookup engine design, able to achieve high-performance with the use of a small portion of a reconfigurable logic device and a commodity Random Access Memory (RAM) device. Based on Eatherton's Tree Bitmap algorithm [1], the Fast Internet Protocol Lookup (FIPL) engine can be scaled to achieve over 9 million lookups per second at the fairly modest clock speed of 100 MHz. FIPL's scalability, efficiency, and favorable update performance make it an ideal candidate for System-On-a-Chip (SOC) solutions for programmable router port processors.**

Keywords— **Internet Protocol (IP) lookup, router, reconfigurable hardware, Field-Programmable Gate Array (FPGA), Random Access Memory (RAM).**

I. Introduction

ROUTING of Internet Protocol (IP) packets is the primary purpose of Internet routers. Simply stated, routing an IP packet involves forwarding each packet along a multi-hop path from source to destination. The speed at which forwarding decisions are made at each router or "hop" places a fundamental limit on the performance of the router. For Internet Protocol Version 4 (IPv4), the forwarding decision is based on a 32-bit destination address carried in each packet's header. A lookup engine at each port of the router uses a suitable routing data structure to determine the appropriate outgoing link for the packet's destination address.

The use of Classless Inter-Domain Routing (CIDR) complicates the lookup process, requiring a lookup engine to search variable-length address prefixes in order to find the longest matching prefix of the destination address and retrieve the corresponding forwarding information [2]. As physical link speeds grow and the number of ports in high-performance routers continues to increase, there is a growing need for efficient lookup algorithms and effective implementations of those algorithms. Next generation routers must be able to support thousands of optical links each operating at 10 Gb/s (OC-192) or more. Lookup techniques that can scale efficiently to high speeds and large lookup table sizes are essential for meeting the growing performance demands while maintaining acceptable per-port costs.

Many techniques are available to perform IP address lookups. Perhaps the most common approach in high-performance systems is to use Content Addressable Memory (CAM) devices and custom Application Specific Integrated Circuits (ASICs). While this approach can provide excellent performance, the performance comes at a fairly high price, due to the relatively high cost per bit of CAMs, relative to commodity memory devices. CAM-based lookup tables are expensive to update, since the insertion of a new routing prefix may require moving an unbounded number of existing entries. The CAM approach also offers little or no flexibility for adapting to new addressing and routing protocols.

The Fast Internet Protocol Lookup (FIPL) engine, developed at Washington University in St. Louis, is a high-performance, solution to the lookup problem, that uses Eatherton's Tree Bitmap algorithm [1], reconfigurable hardware and Random Access Memory (RAM). Implemented in a Xilinx Virtex-E Field Programmable Gate Array (FPGA) running at 100 MHz and using a Micron 1 MB Zero Bus Turnaround (ZBT) Synchronous Random Access Memory (SRAM), a single FIPL lookup engine has a guaranteed worst case performance of 1,134,363 lookups per second. Time-Division Multiplexing (TDM) of eight FIPL engines over a single 36 bit wide SRAM interface, yields a guaranteed worst case performance of 9,090,909 lookups per second. Still higher performance is possible with higher memory bandwidths. In addition, the data structure used by FIPL is straightforward to update, and can support up to 10,000 updates per second with less than a 9% degradation in lookup throughput. Targeted to an open-platform research router, implementations utilized standard FPGA design flows. Ongoing research seeks to exploit new FPGA devices and more advanced CAD tools in order to double the clock frequency and, therefore, double the lookup performance.

Taylor, Lockwood, Sproull, and Turner are with the Applied Research Laboratory, Washington University in Saint Louis. E-mail: {det3,lockwood,todd,jst}@arl.wustl.edu. This work supported in part by NSF ANI-0096052 and Xilinx, Inc.

Parlour is with Xilinx, Inc. E-mail: dave.parlour@xilinx.com

II. Related Work

Numerous research and commercial IP lookup techniques exist. On the commercial front, several companies have developed high speed lookup techniques using CAMs and ASICs. Some current products, targeting OC-768 (40 Gb/s) and quad OC-192 (10 Gb/s) link configurations, claim throughputs of up to 100 million lookups per second and storage for 100 million entries [3]. However, these products requiring 16 cascaded ASICs with embedded CAMs in order to achieve the advertised performance levels as well and to support even a more realistic one million table entries. Such exorbitant hardware resource requirements make these solutions prohibitively expensive for implementation in large routers.

The most efficient lookup algorithm known, from a theoretical perspective is the "binary search over prefix lengths" algorithm described in [4]. The number of steps required by this algorithm grows logarithmically in the length of the address, making it particularly attractive for IPv6, where address lengths increase to 128 bits. However, the algorithm is relatively complex to implement, making it more suitable for software implementation than hardware implementation. It also does not readily support incremental updates.

The Lulea algorithm is the most similar of published algorithms to the Tree Bitmap algorithm used in our FIPL engine [5]. Like Tree Bitmap, the Lulea algorithm uses a type of compressed trie to enable high speed lookup, while maintaining the essential simplicity and easy updatability of elementary binary tries. While similar at a high level, the two algorithms differ in a variety of specifics, that make Tree Bitmap somewhat better suited to efficient hardware implementation.

The remaining sections focus on the design and implementation details of a fast and scalable lookup engine based on the Tree Bitmap algorithm. The FIPL engine offers an efficient and flexible alternative geared to System-On-a-Chip (SOC) router port processor implementations. With tightly bounded worst-case performance and minimal update overhead, FIPL is well-suited for use in high-performance programmable routers, which must be capable of switching even minimum length packets at wire speeds [6].

III. Tree Bitmap Algorithm

Eatherton's Tree Bitmap algorithm is a hardware based algorithm that employs a multibit trie data structure to perform IP forwarding lookups with efficient use of memory [1]. Due to the use of CIDR, a lookup consists of finding the longest matching prefix stored in the forwarding table for a given 32-bit IPv4 destination address and retrieving the associated forwarding information. As shown in Figure 1, the unicast IP address is compared to the stored prefixes starting with the most significant bit. In this example, a packet is bound for a workstation at Washington University in St. Louis. A linear search through the table results in three matching prefixes: *, 10*, and 1000000011*. The third prefix is the longest match, hence its associated forwarding information, denoted by Next Hop 7 in the example, is retrieved. Using this forwarding information, the packet is forwarded to the specified next hop by modifying the packet header.

Prefix	Next Hop
*	35
10*	7
01*	21
110*	9
1011*	1
0001*	68
01011*	51
00110*	3
10001*	6
100001*	33
10000000*	54
1000000000*	12
1000000011*	7

32−bit IP Address
128.252.153.160
1000 0000 1111 1100 ... 1010 0000
Next Hop
7

Fig. 1. IP prefix lookup table of next hops. Next hops for IP packets are found using the longest matching prefix in the table for the unicast destination address of the IP packet.

To efficiently perform this lookup function in hardware, the Tree Bitmap algorithm starts by storing prefixes in a binary trie as shown in 2. Shaded nodes denote a stored prefix. A search is conducted by using the IP address bits to traverse the trie, starting with the most significant bit of the address. To speed up this searching process, multiple bits of the destination address are compared simultaneously. In order to do this, subtrees of the binary trie are combined into single nodes producing a multibit trie; this reduces the number of memory accesses needed to perform a lookup. The depth of the subtrees combined to form a

single multibit trie node is called the stride. An example of a multibit trie using 4-bit strides is shown in Figure 3. In this case, 4-bit nibbles of the destination address are used to traverse the multibit trie. Address_Nibble(0) of the address, 1000_2 in the example, is used for the root node; Address_Nibble(1) of the address, 0000_2 in the example, is used for the next node; etc.

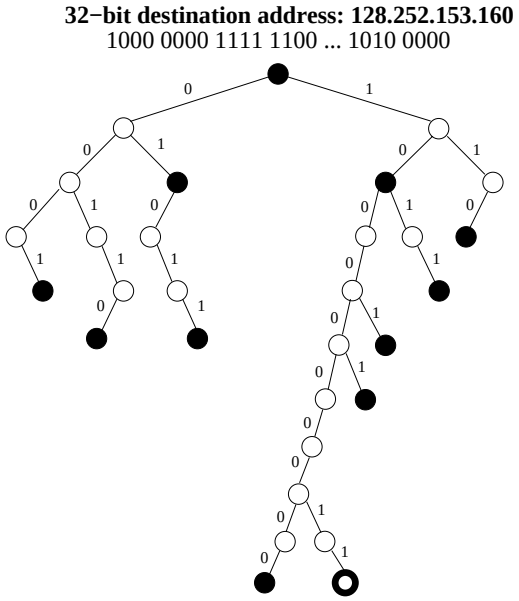

Fig. 2. IP lookup table represented as a binary trie. Stored prefixes are denoted by shaded nodes. Next hops are found by traversing the trie.

The Tree Bitmap algorithm codes information associated with each node of the multibit trie using bitmaps. The *Internal Prefix Bitmap* identifies the stored prefixes in the the binary sub-tree of the multi-bit node. The *Extending Paths Bitmap* identifies the "exit points" of the multibit node that correspond to child nodes. Figure 4 shows how the root node of the example data structure is coded into bitmaps. The 4-bit stride example is shown as a Tree Bitmap data structure in 5. Note that a pointer to the head of the array of child nodes and a pointer to the set of next hop values corresponding to the set of prefixes in the node are stored along with the bitmaps for each node. By requiring that all child nodes of a single parent node be stored contiguously in memory, the address of a child node can be calculated using a single *Child Node Array Pointer* and an index into that array computed from the extending paths bitmap. The same technique is used to find the associated next hop information for a stored prefix in the node. The *Next Hop Table Pointer* points to the beginning of the contiguous set of next hop values corresponding to the set of stored prefixes in the node. Next hop information for a specific prefix may be fetched by indexing from the pointer

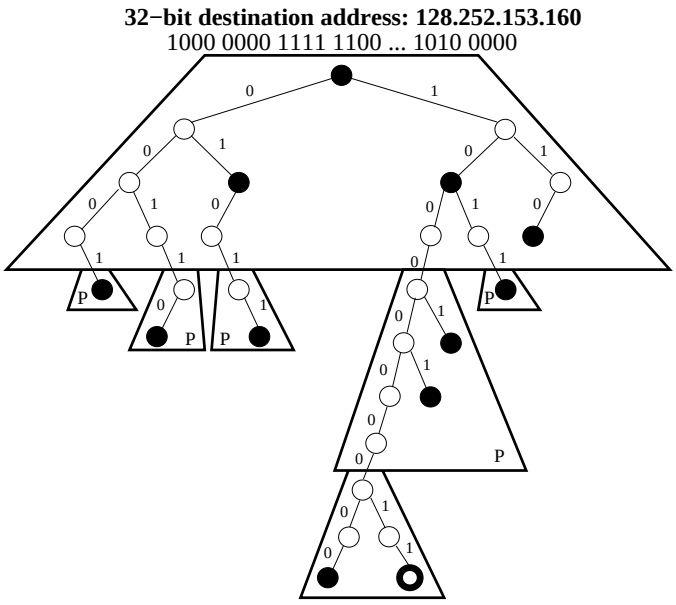

Fig. 3. IP lookup table represented as a multibit trie. A stride, 4-bits, of the unicast destination address of the IP packet are compared at once, speeding up the lookup process.

location.

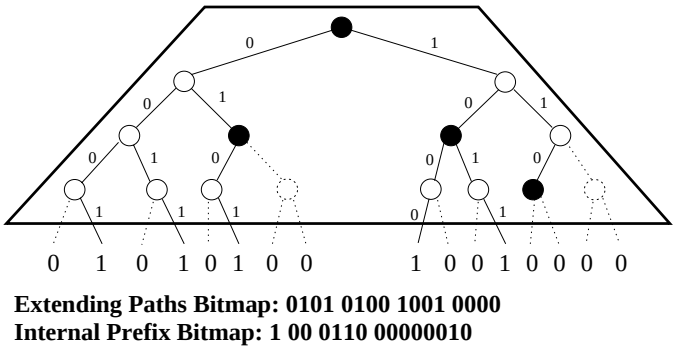

Fig. 4. Bitmap coding of a multibit trie node. The internal bitmap represents the stored prefixes in the node while the extending paths bitmap represents the child nodes of the current node.

The index for the *Child Node Array Pointer* leverages a convenient property of the data structure. Note that the numeric value of the nibble of the the IP address is also the bit position of the extending path in the *Extending Paths Bitmap*. For example, Address_Nibble(0) = 1000_2 = 8. Note that the eighth bit position, counting from the most significant bit, of the *Extending Paths Bitmap* shown in Figure 4 is the extending path bit corresponding to Address_Nibble(0) = 1000_2. The index of the child node is computed by counting the number of ones in the *Extending Paths Bitmap* to the left of this bit position. In the example, the index would be three. This operation of computing the number of ones to the left of a bit position in a bitmap

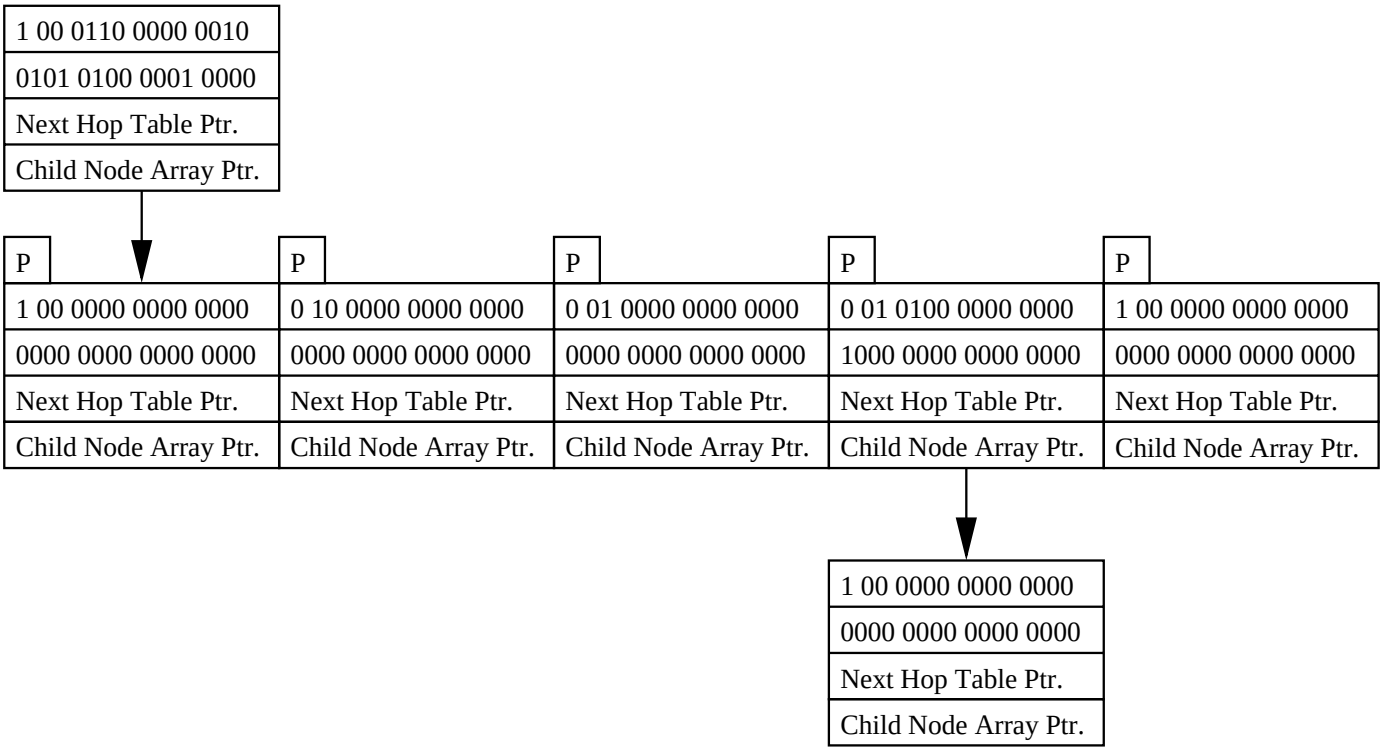

Fig. 5. IP lookup table represented as a Tree Bitmap. Child nodes are stored contiguously so that a single pointer and an index may be used to locate any child node in the the data structure.

will be referred to as *CountOnes* and will be used in later discussions.

When there are no valid extending paths, *Extending Paths Bitmap* is all zeros, the terminal node has been reached and the *Internal Prefix Bitmap* of the node is fetched. A logic operation called *Tree Search* returns the bit position of the longest matching prefix in the *Internal Prefix Bitmap*. *CountOnes* is then used to compute an index for the *Next Hop Table Pointer.*, and the next hop information is fetched. If there are no matching prefixes in the *Internal Prefix Bitmap* of the terminal node, then the *Internal Prefix Bitmap* of the most recently visited node that contains a matching prefix is fetched. This node is identified using a data structure optimization called the *Prefix Bit*.

The *Prefix Bit* of a node is set if its parent has any stored prefixes along the path to itself. When searching the data structure, the address of the last node visited is remembered. If the current node's *Prefix Bit* is set, then the address of the last node visited is stored as the best matching node. Setting of the *Prefix Bit* in the example data structure of Figure 3 and Figure 5 is denoted by a "P".

IV. Hardware Design and Implementation

Modular design techniques are employed throughout the FIPL hardware design to provide scalability for various system configurations. Figure 6 details the components required to implement FIPL in the Port Processor (PP) of a router. Other components of the router include the Transmission Interfaces (TI), Switch Fabric, and Control Processor (CP). Providing the foundation of the FIPL design, the FIPL engine implements a single instance of a Tree Bitmap search. The FIPL Engine Controller may be configured to instantiate multiple FIPL engines in order to scale the lookup throughput with system demands. The FIPL Wrapper extracts the IP addresses from incoming packets and writes them to an address FIFO read by the FIPL Engine Controller. Lookup results are written to a FIFO read by the FIPL Wrapper which accordingly modifies the packet header. The FIPL Wrapper also handles standard IP processing functions such as checksums and header field updates. Specifics of the FIPL Wrapper will vary depending upon the type of switching core and transmission format. An on-chip Control Processor receives and processes memory update commands on a dedicated control channel. Memory updates are the result of route add, delete, or modify commands and are sent from the System Management and Control components. Note that the off-chip memory is assumed to be a single port device; hence, an SRAM Interface arbitrates access between the FIPL Engine Controller and Control Processor.

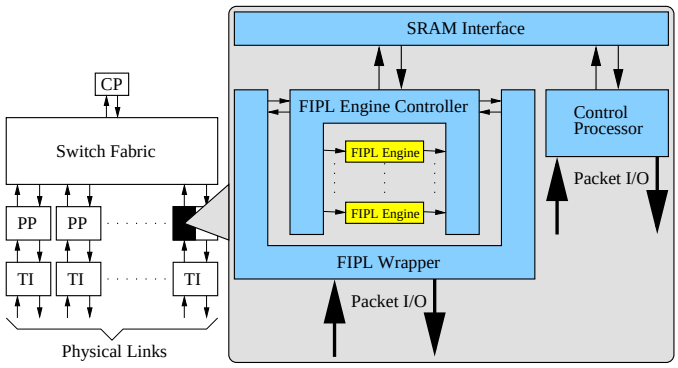

Fig. 6. Block diagram of router with multi-engine FIPL configuration; detail of FIPL system components in the Port Processor (PP).

A. FIPL Engine

Consisting of a few address registers, a simple Finite-State Machine (FSM), and combinational logic, the FIPL Engine is a compact, efficient Tree Bitmap search engine. A dataflow diagram of the FIPL Engine is shown in Figure 7. Data arriving from memory is latched into the DATA_IN_REG register *n* clock cycles after issuing a memory read. The value of *n* is determined by the read latency of the memory device plus 2 clock cycles for latching the address out of and the data into the implementation device. The next address issued to memory is latched into the ADDR_OUT_REG *k* clock cycles after data arrives from memory. The value of *k* is determined by the speed at which the implementation device can compute the *next_hop_addr* which is the critical path in the logic. Two counters, *mem_count* and *search_count*, are used to count the number of clock cycles for memory access and address calculation, respectively. Use of multicycle paths allows the FIPL engine to scale with implementation device and memory device speeds by simply changing compare values in the finite-state machine logic.

In order to generate *next_hop_addr*:

- TREE_SEARCH generates *prefix_index* which is the bit position of the best-matching prefix stored in the *Internal Prefixes Bitmap*
- PREFIX_COUNTONES generates *next_hop_index* which is the number of 1's to the left of *prefix_index* in the *Internal Prefixes Bitmap*
- *next_hop_index* is added to the lower four bits of the *Next Hop Table Pointer*
- The carryout of the previous addition is used to select the upper bits of the *Next Hop Table Pointer* or the precomputed value of the upper bits plus 1

The NODE_COUNTONES and identical fast addition blocks generate the *child_node_addr*, but require less time as the TREE_SEARCH block is not in the path. The ADDR_OUT_MUX selects the next address issued to memory among the addresses for the next root node's *Extending Paths Bitmap* and *Child Node Array Pointer* (*root_node_ptr*), the next child node's *Extending Paths Bitmap* and *Child Node Array Pointer* (*child_node_addr*), the current node's *Internal Prefix Bitmap* and *Next Hop Table Pointer* (*curr_node_prefixes_addr*), the forwarding information for the best-matching prefix (*next_hop_addr*), and the best-matching previous node's *Internal Prefix Bitmap* and *Next Hop Table Pointer* (*bestmatch_prefixes_addr*). Selection is made based upon the current state.

VALID_CHILD examines the *Extending Paths Bitmap* and determines if a child node exists for the current node based on the current nibble of the IP address. The output of VALID_CHILD, *prefix_index*, *mem_count*, and *search_count* determine state transitions as shown in Figure 8. The current state and the value of the P_BIT determine the register enables for the BESTMATCH_PREFIXES_ADDR_REG and the BESTMATCH_STRIDE_REG which store the address of the *Internal Prefixes Bitmap* and *Next Hop Table Pointer* of the node containing best-matching prefixes and the associated stride of the IP address, respectively.

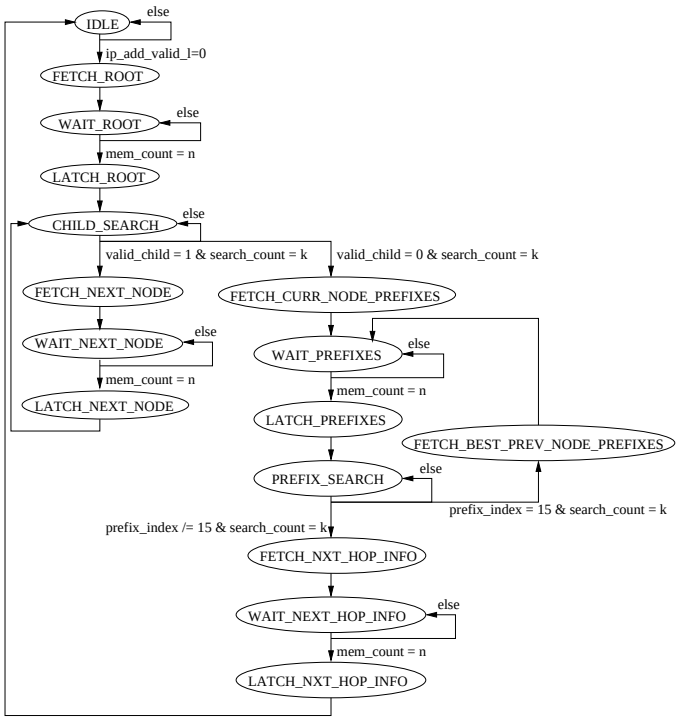

Fig. 8. FIPL engine finite-state-machine bubble diagram.

B. FIPL Engine Controller

Leveraging the uniform memory access period of the FIPL Engine, the FIPL Engine Controller employs a sim-

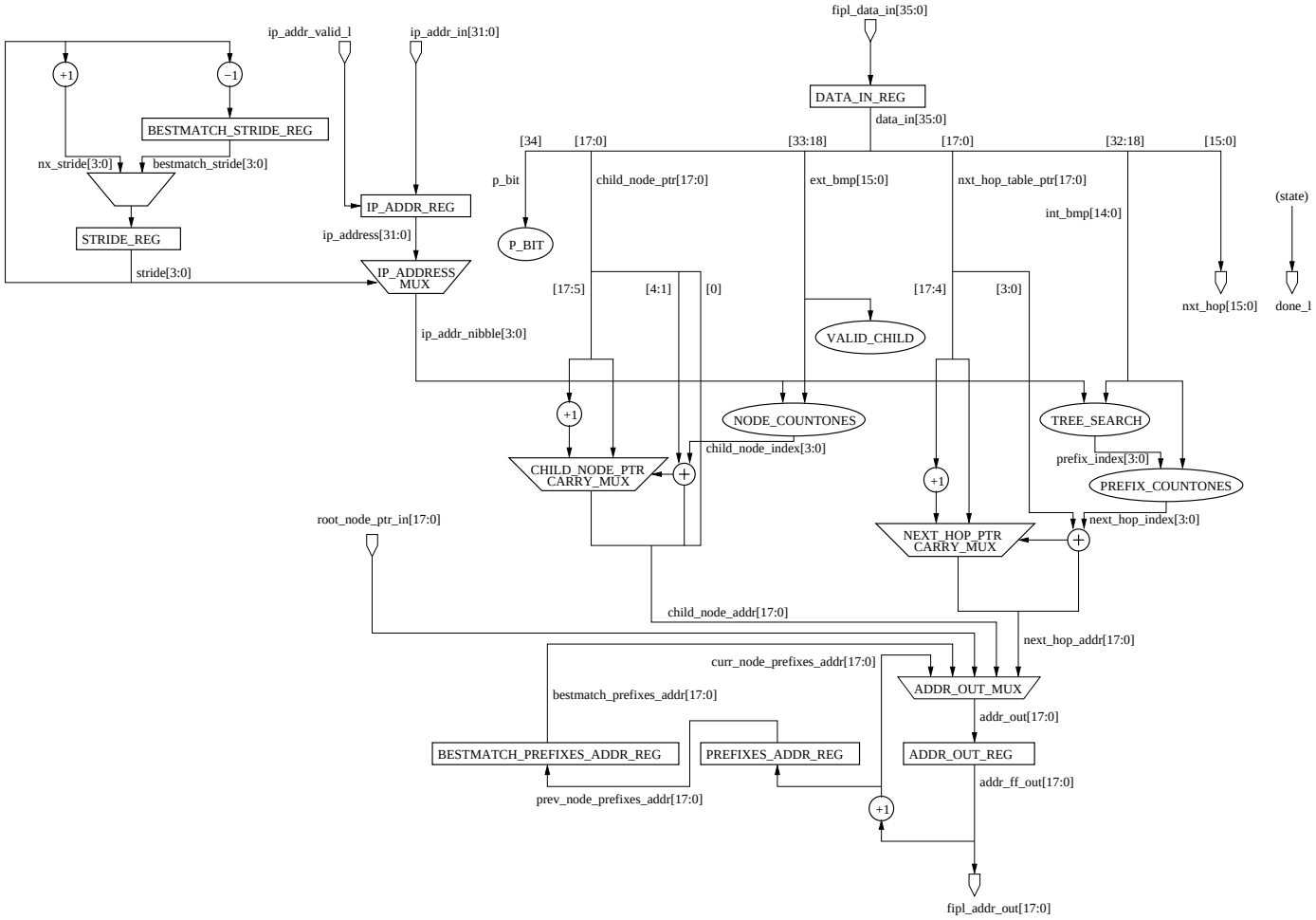

Fig. 7. FIPL engine dataflow; multi-cycle path from DATA_IN_FLOPS to ADDR_OUT_FLOPS can be scaled according to target device speed; all multiplexor select lines and flip-flop enables implicitly driven by finite-state machine outputs.

ple Time Division Multiplexing (TDM) design to scale lookup throughput in order to meet system demands. The scheme centers around a timing wheel with a number of slots equal to the FIPL Engine memory access period. When an address is read from the input FIFO, the next available FIPL Engine is started at the next available time slot. The next available time slot is determined by indexing the current slot time by the known startup latency of a FIPL Engine. For example, assume an access period of 8 clock cycles; hence, the timing wheel has 8 slots numbered 0 through 7. Assume three FIPL Engines are currently performing lookups occupying slots 1, 3, and 4. Furthermore, assume that from the time the IP address is issued to the FIPL Engine to the time the FIPL Engine issues its first memory read is 2 clock cycles; hence, the startup latency is 2 slots. When a new IP address arrives, the next lookup may not be started at slot times 7, 1, or 2 because the first memory read would be issued at slot time 1, 3, or 4, respectively which would interfere with ongoing lookups. Assume the current slot time is 3; therefore, the next FIPL engine is started and slot 5 is marked as occupied.

As previously mentioned, input IP addresses and output forwarding information are passed between the FIPL Engine Controller and the FIPL Wrapper via FIFO interfaces. This design simplifies the design of the FIPL Wrapper by placing the burden of in-order delivery of results on the FIPL Engine Controller. While individual input and output FIFOs could be used for each engine to prevent head-of-the-line blocking, network designers will usually choose to configure the FIPL Engine Controller assuming worst-case lookups. Also, the performance numbers reported in a subsequent section show that average lookup latency per FIPL Engine increases by less than 6% for an 8-engine configuration; therefore, lookup engine “dead-time” is negligible.

C. Implementation Platform

FIPL is implemented on open-platform research systems designed and built at Washington University in Saint Louis [7]. The WUGS 20, an 8-port ATM switch providing 20 Gb/s of aggregate throughput, provides a high-

performance switching fabric [8]. This switching core is based upon a multi-stage Benes topology, supports up to 2.4 Gb/s link rates, and scales up to 4096 ports for an aggregate throughput of 9.8 Tb/s [9]. Each port of the WUGS 20 can be fitted with a Field Programmable Port Extender (FPX), a port card of the same form factor as the WUGS transmission interface cards [10]. Each FPX contains two FPGAs, one acting as the Network Interface Device (NID) and the other as the Reprogrammable Application Device (RAD). The RAD FPGA has access to two 1MB Zero Bus Turnaround (ZBT) SRAMs and two 64MB SDRAM modules providing a flexible platform for implementing high-performance networking applications [11].

To allow for packet reassembly and other processing functions requiring memory resources, the FIPL has access to one of the 8 Mbit ZBT (Zero Bus Turnaround) SRAMs which require 18-bit addresses and provide a 36-bit data path with a 2-clock cycle latency. Since this memory is ”off-chip” both the address and data lines must be latched at the pads of the FPGA, providing for a total latency to memory of n = 4 clock cycles. Utilizing a 4-bit stride the *Extending Paths Bitmap* is 16-bits long, occupying less than a half-word of memory. The remaining 20-bits of the word are used for the *Prefix Bit* and *Child Node Array Pointer*; hence, only one memory access is required per node when searching for the terminal node. Likewise, the *Internal Prefix Bitmap* and *Next Hop Table Pointer* may be stored in a single 36-bit word; hence, a single node of the Tree Bitmap requires two words of memory space. 131,072 nodes may be stored in one of the 8Mbit SRAMs providing a maximum of 1,966,080 stored routes.

In this configuration, the pathological lookup requires 11 memory accesses: 8 memory accesses to reach the terminal node, 1 memory access to search the sub-tree of the terminal node, 1 memory access to search the sub-tree of the most recent node containing a match, and 1 memory access to fetch the forwarding information associated with the best-matching prefix. Since the FPGAs and SRAMs run on a synchronous 100MHz clock, all single cycle calculations must be completed in less than 10ns. The critical path in the FIPL design, resolving the *next_hop_addr*, requires more than 20 ns when targeted to the RAD FPGA of the FPX, a Xilinx XCV1000E-7; hence, k is set to 3. This provides a total memory access period of 80 ns and requires 8 FIPL engines in order to fully utilize the available memory bandwidth. Theoretical worst-case performance, all lookups requiring 11 memory accesses, ranges from 1,136,363 lookups per second for a single FIPL engine to 9,090,909 lookups per second for eight FIPL engines in this implementation environment.

As the WUGS 20 supports a maximum line speed of 2.4 Gb/s, a 4-engine configuration is used in the Washington University system. Due to the ATM switching core, the FIPL Wrapper supports AAL5 encapsulation of IP packets inside of ATM cells [12]. Relative to the Xilinx Virtex 1000E FPGA used in the FPX, each FIPL Engine utilizes less than 1% of the available logic resources. Configured with 4 FIPL Engines, FIPL Engine Controller utilizes approximately 6% of the logic resources while the FIPL Wrapper utilizes another 2% of the logic resources and 12.5% of the on-chip memory resources. This results in an 8% total logic resource consumption by FIPL. The SRAM Interface and Control Processor which parses control cells and executes memory commands for route updates utilize another 8% of the available logic resources and 2% of the on-chip memory resources. Therefore, all input IP forwarding functions occupy 16% of the logic resources leaving the remaining 74% of the device available for other packet processing functionality.

V. System Management and Control Components

System management and control of FIPL in the Washington University system is performed by several distributed components. All components were developed to facilitate further research using the open-platform system.

A. NCHARGE

NCHARGE is the software component that controls reprogrammable hardware on a switch. Figure 9 shows the role of NCHARGE in conjunction with multiple FPX devices within a switch. The software provides connectivity between each FPX and multiple remote software processes via TCP sockets that listen on a well-defined port. Through this port, other software components are able to communicate to the FPX using its specified API. Because each FPX is controlled by an independent NCHARGE software process, distributed management of entire systems can be performed by collecting data from multiple NCHARGE elements. [13].

B. FIPL Memory Manager

The FIPL Memory Manager is a stand alone C++ application that accepts commands to add, delete, and update routing entries for a hardware-based Internet router. The program maintains the previously discussed Tree Bitmap data structure in a shared memory between hardware and software . When a user enters route updates, the FIPL Memory Manager Software returns the corresponding memory updates needed to perform that operation in the FPX hardware.

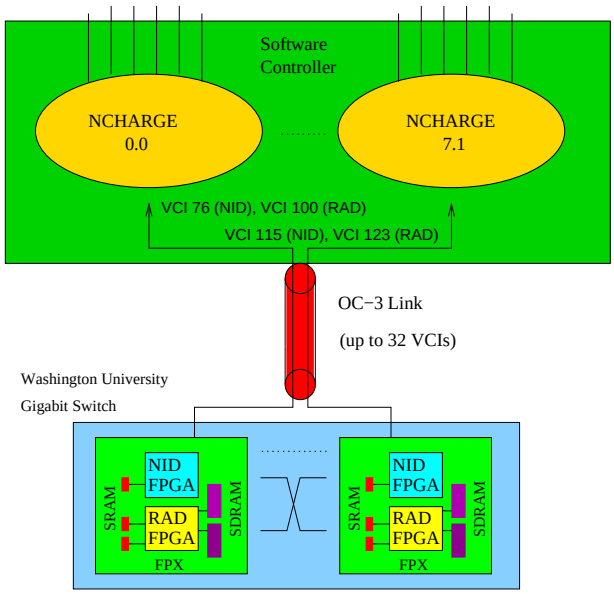

Fig. 9. Detail of the hardware and software components that comprise the FPX system. Each FPX is controlled by an NCHARGE software process. The contents of the memories on the FPX modules can be modified by remote processes via the software API to NCHARGE.

```
Command options:
        [A]dd
        [D]elete
        [C]hange
        [P]rint
        [M]emoryDump
        [Q]uit

Enter command (h for help): A
You entered add

Enter prefix x.x.x.x/s
(x = 0-255, s is significant bits 0-32)
192.128.1.1/8

Enter Next Hop value: 4
******
Memory Update Commands:

w36 0 4 2 000000000 100000006
w36 0 2 2 200000004 000000000
w36 0 0 2 000200002 000000000
```

In the example shown here a single add route command requires three 36-bit memory write commands, each consisting of 2 consecutive locations in memory at addresses 4, 2, and 0, respectively.

C. Sockets Interfaces

In order to access the FIPL Memory Manager as a daemon process, support software needs to be in place to handle standard input and output. Socket software was developed to handle incoming route updates to pass along to the FIPL Memory Manager. A socket interface was also developed to send the resulting output of a memory update to the NCHARGE software. These software processes handling input and output are called Write_Fip and Read_Fip, respectively. Write_Fip is constantly listening on a well known port for incoming route update commands. Once a connection is established the update command is sent as an ASCII character string to Write_Fip. This software prints the string as standard output which is redirected to the standard input of FIPL Memory Manager. The memory update commands needed by NCHARGE software to perform the route update are issued at the output of FIPL Memory Manager. Read_Fip receives these commands as standard input and sends all of the memory updates associated with one route update over a TCP socket to the NCHARGE software.

D. Remote User Interface

The current interface for performing route updates is via a web page that provides a simple interface for user interaction. The user is able to submit single route updates or a batch job of multiple routes in a file. Another option available to users is the ability to define unique control cells. This is done through the use of software modules that are loaded into the NCHARGE system.

In the current FIPL Module, a web page has been designed to provide a simple interface for issuing FIPL control commands, such as changing the *Root Node Pointer*. The web page also provides access to a vast database of sample route table entries taken from the Internet Performance Measurement and Analysis project's website [14]. This website provides daily snapshots of Internet backbone routing tables including traditional Class A, B, and C addresses. Selecting the download option from the FIPL web page executes a Perl script to fetch the router snapshots from the database. The Perl script then parses the files and generates an output file that is readable by the Fast IP Lookup Memory Manager.

E. Command Flow

The overall flow of data with FIPL and NCHARGE is shown in Figure 10. Suppose a user wishes to add a route to the database. The user first submits either a single command or submits a file containing multiple route updates. Data submitted from the web page, Figure 11, is passed

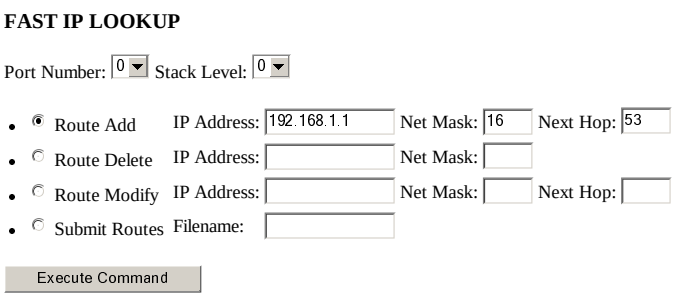

Fig. 11. FPX Web Interface for FIPL route updates.

to the Web Server as a form. Local scripts process the form and generate an Add Route command that the software understands. These commands are ASCII strings in the form "Add route $A_1.A_2.A_3.A_4$/netmask nexthop. The script then sets up a TCP Socket and transmits each command to the Write_Fip software process. As mentioned before Write_fip listens on a TCP port and relays messages to standard output in order to communicate with the FIPL Memory Manager. FIPL Memory Manager takes the standard input and processes the route command in order to generate memory updates for an FPX board. Each memory update is then passed as standard output to the Read_Fip process.

After this process collects memory updates it establishes a TCP connection with NCHARGE to transmit the commands. Read_Fip is able to detect individual route commands and issues the set of memory updates associated with each. This prevents Read_Fip from creating a socket for every memory update. From here memory updates are sent to NCHARGE software process to be packed into control cells to send to the FPX. NCHARGE packs as many memory commands as it can fit into a 53 byte ATM cell while preserving order between commands. NCHARGE sends these control cells using a stop-and-wait protocol to ensure correctness, then issues a response message to the user.

VI. Performance

While the worst-case performance of FIPL is deterministic, an evaluation environment was developed in order to benchmark average FIPL performance on actual router databases. As shown in Figure 12, the evaluation environment includes a modified FIPL Engine Controller, 8 FIPL Engines, and a FIPL Evaluation Wrapper. The FIPL Evaluation Wrapper includes an IP Address Generator which uses 16 of the available on-chip BlockRAMs in the Xilinx Virtex 1000E to implement storage for 2048 IPv4 destination addresses. The IP Address Generator interfaces to the FIPL Engine controller like a FIFO. When a test run is initiated, an empty flag is driven to FALSE until all 2048

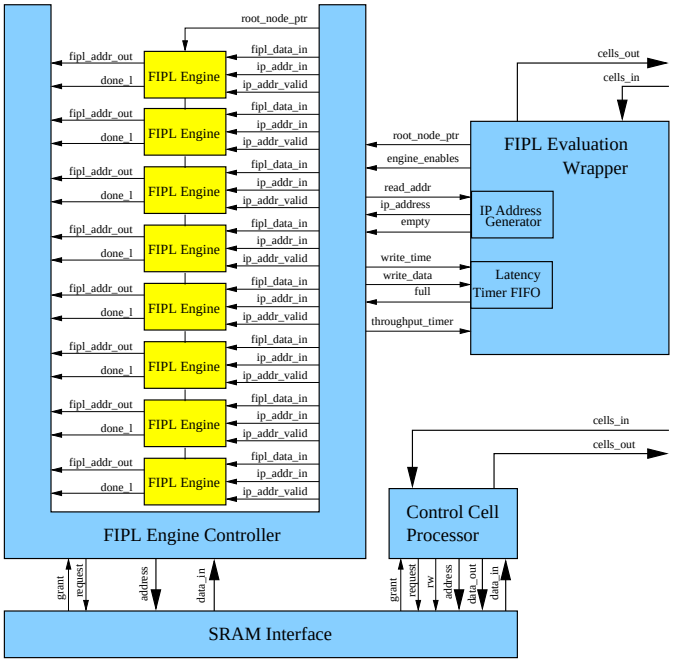

Fig. 12. Block diagram of FIPL evaluation environment.

addresses are read.

Control cells sent to the FIPL Evaluation Wrapper initiate test runs of 2048 lookups and specify how many FIPL Engines should be used during the test run. The FIPL Engine Controller contains a latency timer for each FIPL Engine and a throughput timer that measures the time required to complete the test run. Latency timer values are written to a FIFO upon completion of each lookup. The FIPL Evaluation Wrapper packs latency timer values into control cells which are sent back to the system control software where the contents are dumped to a file. The throughput timer value is included in the final control cell.

Using a portion of the Mae-West snapshot from July 12, 2001, a Tree Bitmap data structure consisting of 16,564 routes was loaded into the off-chip SRAM. The on-chip memory read by the IP Address Generator was initialized with 2048 destination addresses randomly selected from the route table snapshot. Test runs were initiated using 1 through 8 engines. Figure 13 shows the results of test runs without intervening update traffic. Plots of the theoretical performance for all worst-case lookups is shown for reference. Figure 14 shows the results of test runs with various intervening update frequency. An update consisted of a route addition requiring 12 memory writes packed into 3 control cells.

With no intervening update traffic, lookup throughput ranged from 1,526,404 lookups per second for a single FIPL engine to 10,105,148 lookups per second for 8 FIPL engines. Average lookup latency ranged from 624 ns for a

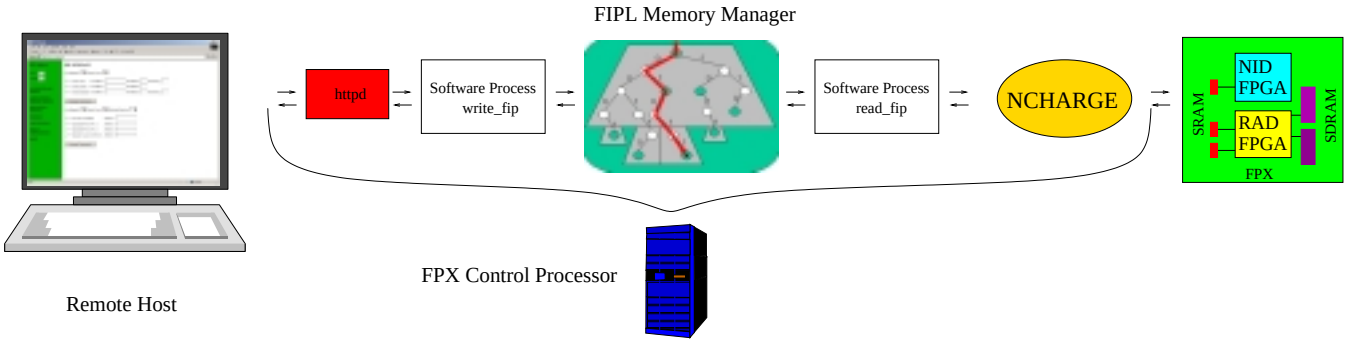

Fig. 10. Data flow with FIPL and NCHARGE

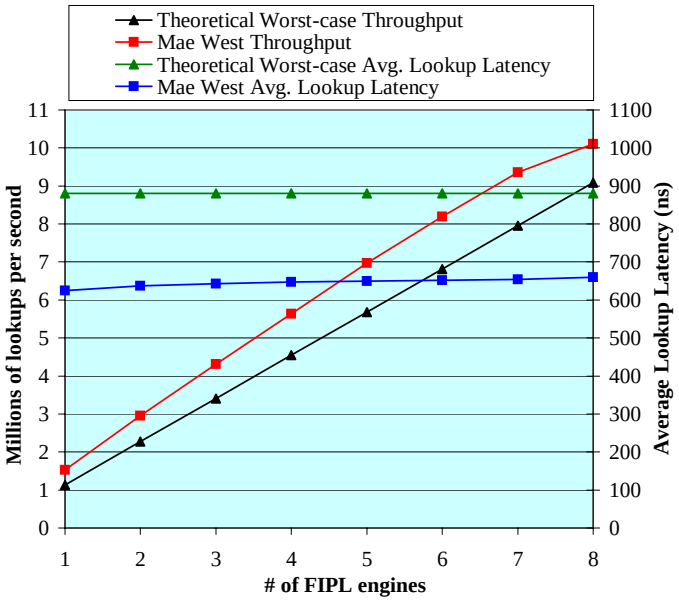

Fig. 13. FIPL performance: measurements used a sample database from Mae West on July 12, 2001 consisting of 16,564 routes. Input test vectors consisted of random selections of 2048 IPv4 destination addresses.

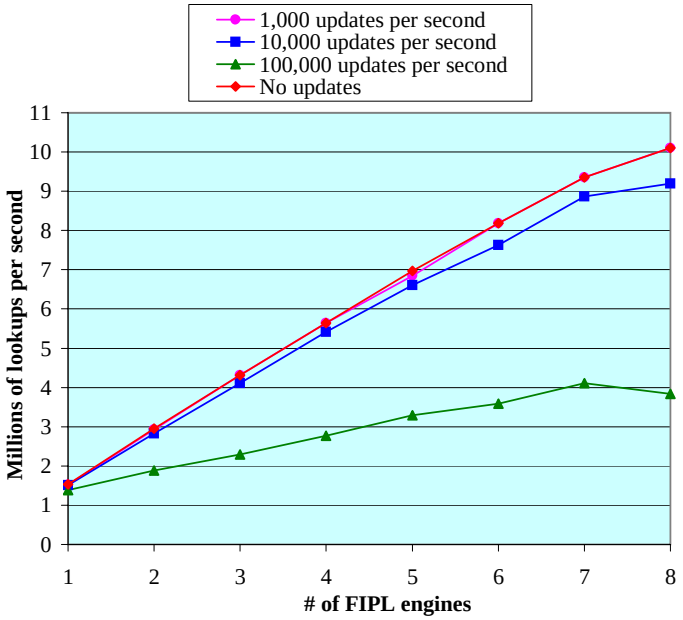

Fig. 14. FIPL performance under update load: measurements used a sample database from Mae West on July 12, 2001 consisting of 16,564 routes. Input test vectors consisted of random selections of 2048 IPv4 destination addresses. A single update consisted of a route addition requiring 12 memory writes packed into 3 control cells.

single FIPL engine to 660 ns for 8 FIPL engines. This is less than a 6% increase in average lookup latency over the range of FIPL Engine Controller configurations.

Note that update frequencies up to 1,000 updates per second have little to no effect on lookup throughput performance. An update frequency of 10,000 updates per second exhibited a maximum performance degradation of 9%. Using the near maximum update frequency supported by the Control Processor of 100,000 updates per second, lookup throughput performance is degraded by a maximum of 62%. Note that this is a highly unrealistic situation, as lookup frequencies rarely exceed 1,000 updates per second.

VII. Ongoing Research

Coupled with advances in FPGA device technology, implementation optimizations of critical paths in the FIPL engine circuit hold promise of doubling the system clock frequency to 200 MHz in order to take full advantage of the memory bandwidth offered by the ZBT SRAMs. Doubling of the clock frequency directly translates to a doubling of the lookup performance to a guaranteed worst case throughput of over 18 million lookups per second.

The *CountOnes* operation can be accelerated by replacing the current multi-level logic implementation with a table lookup tailored to the specific resources available on the FPGA. The Virtex FPGA provides columns of dual-ported 4096 bit BlockRAMs, which can be configured to

various sizes. Two BlockRAMs in a 2048 x 2 organization, whose contents are initialized by the FPGA's configuration bitstream, can be combined to act as a dual-ported 2048 x 4 Read Only Memory (ROM). In addition, the BlockRAMs feature a registered output with synchronous reset which facilitates pipelining. A single ROM can perform the *CountOnes* table lookup on the lower 8 bits and upper 8 bits of a 16-bit bitmap simultaneously, since each address port has 11 bits (8 bits for the bitmap value and 3 bits for selecting the number of bit positions to be counted). The lower and upper count values must then be added to the base pointer, either the *Next Hop Table Pointer* or the *Child Array Pointer*, to determine the address of the next memory location to be read. A single level of logic is required at the ROM address inputs to force all 8 lower bits to be counted for 4-bit stride values of 8 or more. The output register resets are used to force all outputs to zero when the stride value is zero and the upper count value to zero when the stride value is 8 or less.

Experiments with this FPGA-specific implementation of the *Countones* operation have shown that, with appropriate pipelining at the BlockRAM address inputs as well as the output additions, operation in excess of 100 MHz with no multicycle paths is feasible. This means that two engines built in this fashion could fully utilize the available bandwidth of a ZBT SRAM running at 200 MHz.

VIII. Conclusions

As optical link speeds continue to increase demands for performance and embedded network services impose flexibility demands, Internet routers must become more efficient and programmable. IP address lookup is one of the primary functions of the router and often is a significant performance bottleneck. Fast Internet Protocol Lookup (FIPL) utilizes Eatherton's Tree Bitmap algorithm, reconfigurable hardware, and Random Access Memory (RAM) to implement a scalable, high-performance IP lookup engine capable of at least 9 million lookups per second. Utilizing only a fraction of a reconfigurable logic device and a single RAM device, FIPL offers an attractive alternative to expensive commercial solutions employing multiple Content Addressable Memory (CAM) devices and Application Specific Integrated Circuits (ASICs). By providing high-performance at low per-port costs, FIPL is a prime candidate for System-On-a-Chip (SOC) solutions for next generation programmable router port processors.

References

[1] W. N. Eatherton, "Hardware-Based Internet Protocol Prefix Lookups," thesis, Washington University in St. Louis, 1998.

[2] S. Fuller, T. Li, J. Yu, and K. Varadhan, "Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation," Internet RFC 1519, Sept. 1993.

[3] SiberCore Technologies Inc., "SiberCAM Ultra-2M SCT2000 Product Brief," 2000.

[4] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner, "Scalable high speed IP routing table lookups," in *Proceedings of ACM SIGCOMM '97*, September 1997, pp. 25–36.

[5] Andrej Brodnik, Svante Carlsson, Mikael Degermark, and Stephen Pink, "Small Forwarding Tables for Fast Routing Lookups," in *SIGCOMM 97*, 1997, pp. 3–14.

[6] David E. Taylor, Jon S. Turner, and John W. Lockwood, "Dynamic Hardware Plugins (DHP): Exploiting reconfigurable hardware for high-performance programmable routers," in *IEEE OPENARCH 2001: 4th IEEE Conference on Open Architectures and Network Programming*, Anchorage, AK, Apr. 2001.

[7] Jonathan S. Turner, "Gigabit Technology Distribution Program," http://www.arl.wustl.edu/gigabitkits/kits.html, Aug. 1999.

[8] J. Turner, T. Chaney, A. Fingerhut, and M. Flucke, "Design of a Gigabit ATM Switch," in *In Proceedings of Infocom 97*, Mar. 1997.

[9] Sumi Choi, John Dehart, Ralph Keller, John W. Lockwood, Jonathan Turner, and Tilman Wolf, "Design of a flexible open platform for high performance active networks," in *Allerton Conference*, Champaign, IL, 1999.

[10] John W. Lockwood, Jon S. Turner, and David E. Taylor, "Field programmable port extender (FPX) for distributed routing and queuing," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, Monterey, CA, USA, Feb. 2000, pp. 137–144.

[11] John W. Lockwood, Naji Naufel, Jon S. Turner, and David E. Taylor, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, Monterey, CA, USA, Feb. 2001, pp. 87–93.

[12] Peter Newman et al., "Transmission of flow labelled IPv4 on ATM data links," Internet RFC 1954, May 1996.

[13] James M. Anderson, Mohammad Ilyas, and Sam Hsu, "Distributed network management in an internet environment," in *Globecom'97*, Pheonix, AZ, Nov. 1997, vol. 1, pp. 180–184.

[14] "Internet Routing Table Statistics," http://www.merit.edu/ipma-/routing_table/, May 2001.