DesignCon 2005

Hardware Implementation of a Tree Based IP Lookup Algorithm for OC-768 and beyond

Florin Baboescu, ST Microelectronics Inc.
Suresh Rajgopal, ST Microelectronics Inc.
Lun-Bin Huang, ST Microelectronics Inc.
Nick Richardson, ST Microelectronics Inc.

Abstract

Continuing growth in link speeds and the number of advertised IP prefixes places increasing demands on the performance of Internet routers. An Internet Service Provider (ISP) requires routers to be able to accommodate up to 500,000 prefixes and very likely 1M in the near future. Also the prefix length is expected to grow from (up to) 32 bits to 128 bits with the introduction of IPv6.

While ternary-CAM based lookup solutions serve the needs of today's routers, they do not scale well for the next-generation. TCAMs are expensive and consume a lot of power. Algorithmic solutions amenable to software implementation are scalable and cost-effective, but cannot deliver the throughput needed by OC-768 links. A pipelined hardware implementation of an algorithmic solution can deliver high throughput, store a large number of entries with wide entry lengths and is cost-effective and scalable.

This paper describes single chip pipeline architecture for tree based searches that can support IP lookups at a rate of up to 250M searches/s on routing tables with up to 512K entries (either IPv4 or IPv6). It is used in conjunction with a multi-bit trie based algorithmic solution.The architecture relies on an embedded CPU to offload NPU table management functions. On chip firmware allows fast incremental prefix updates, without affecting the search rate. The update rate is at least an order of magnitude better than that required by the IP protocols.

Our solution has separate 36-bit QDRII interfaces to connect to the NPU and to an external SRAM (for next hop data). The two high-speed interfaces running at 250MHz rely on source-synchronous DLL clocking to deliver 500Mbits/sec/pin. Internally, it relies on a deeply pipelined search path to deliver search throughput of up to 250Msps. In addition, to meet the memory access needs of the algorithm an innovative highly configurable embedded SRAM array was designed. It is able to perform up to 4 billion reads/second, with an internal bandwidth of 40GB/s. It also offers extremely high reliability with ST's patented Robust SRAM process to reduce Soft Error rate. In conjunction with our multi-bit trie algorithm for IP lookups it offers a capacity similar to that of an 18Mbit TCAM at a much lower cost and 1/6 the power of the TCAM.

The paper is organized as follows. It first introduces the IP lookup problem and presents a brief background on algorithmic solutions. It also contrasts the use of algorithmic solutions against TCAM based solutions and then describes our solution based on a compressed representation of a multi-bit trie based search structure. Then challenges of a hardware implementation of the algorithm (to meet OC-768 requirements) are described along with our solution and architecture. The role of the embedded CPU in memory allocation and memory management during pipelined tree based searches is described and the paper highlights our firmware solution for dynamic memory allocation and the design of an efficient low cost memory manager. We conclude with a description of the design, modeling and verification methodology and share experiences that may be used in future complex SoC designs.

Deleted: more

Deleted: a single

Authors' Biographies

Nick Richardson is a Fellow at STMicroelectronics, and the manager of the Central R&D Advanced Designs Group in San Diego. He has an extensive background in definition and design of CPU micro-architectures, cache controllers, and I/O subsystems and holds over 15 US patents. He graduated from the University of London.

Lun bin Huang is a Sr. Principal Engineer at the ST Microelectronics Central R&D Advanced Designs Group in San Diego. He has over 15 yrs experience in the areas of architecture, micro-architecture, modeling, design, and implementation of digital devices. He has a BSEE in Computer Science and Engineering and an MSEE degree in Communications from California State University, Long Beach.

Suresh Rajgopal is a Principal Engineer the ST Microelectronics Central R&D Advanced Designs Group in San Diego. He has over 11 yrs of experience working in the areas of design automation, low-power design, SoC and CPU architecture, design and modeling. He has a Ph.D. in Computer Science and Engineering from the University of North Carolina at Chapel Hill.

Florin Baboescu has been with STMicroelectronics since the summer of 2002. His area of expertise is Computer Networks, Computer Architectures and Operating Systems. In Computer Networks he developed algorithms for IP lookups/classification, routing and scheduling while in the field of Computer Architecture he was one of the first to develop a memory system for a Simultaneous Multithreading (SMT) system. He has a PhD in Computer Science and Engineering from the Univ. of California, San Diego.

1 Introduction

The rapid growth of the Internet has brought great challenges in deploying high-speed networks. One particular challenge is to provide high packet forwarding rates through the router. Network search engines capable of providing IP lookup, VPN forwarding, or packet classification are a major component of every router. With the increase in link speeds, increase in the number of advertised IP prefixes, and deployment of new network services the demands placed on these network search engines are increasingly causing them to become a potential bottleneck for the router. According to a recent survey from Linley Group the search engine market grew a healthy 14% from 83 million USD in 2002 to 95 million USD in 2003 with the potential of exceeding 200 million USD in 2007.

Most of the search engine designs use either dedicated ASICs or Ternary CAMs. Hardware based solutions based on Ternary CAMs [1] have been the most popular implementation in routers, due to their efficiency. TCAMs are content addressable memories in which each bit is allowed to store a *0*, *1* or a ``*don't care*'' value. A TCAM compares each packet address with every address the search engine holds in its database, using parallel lookups on associative memory. However TCAMs have their limitations: **(1)** large cell size (about 16 transistors per bit), **(2)** significant high power consumption (a 18Mbits TCAM consumes about 15W at 133Msps), **(3)** very high cost ($300 for a 18Mbits TCAM) and **(4)** can not provide an efficient scalable single chip solution for a search that requires storing more than 128,000 IPv6 prefixes.

Deleted: provide an attractive alternative solution to ASIC-based designs that implement tree based algorithmic solutions for searches.

Deleted: essentially

Deleted: to a

Algorithmic based solutions are an attractive alternative to overcome these limitations. They are scalable, low-power, cost-effective and amenable to software implementation. But they suffer from their own problems, *viz.* long latencies, unpredictable capacity due to search sensitivity and an inability to keep up with the ever-increasing lookup rates demanded by NPUs.

Deleted: ,

Deleted: as we shall see,

Most algorithmic-based solutions for network searches can be regarded as some form of tree traversal, where the search starts at the root node, traverses various levels of the tree, and typically ends at a leaf node. For example, most Internet packets require a longest matching prefix of a 32-bit destination address in a prefix table of about several hundred thousands prefixes. The most common data structure for doing prefix lookups is some form of trie, where a trie is a tree where branching decisions are made based on values of successive bits in the destination address. Using a single computational logic that we call a processing block, even the fastest of these lookup schemes take at least 8 memory accesses per lookup. As link speeds scale rapidly to OC-768, a packet look-up is required every 4 ns. With memory speeds increasing very slowly in comparison, IP lookup will soon become a bottleneck for core routers.

Fortunately, every tree based lookup scheme can be pipelined using several processing blocks. Part of the lookup for the first packet is done by the first block in the pipeline, and then the packet data is passed to the second processing block while the first

block works on the second packet. Thus several modern routers use algorithmic solutions that perform one lookup every memory access after pipeline fills. In this way one can handle OC-768 using pipelined forwarding engine. This also makes the solution amenable to hardware implementation. However, although the idea sounds simple, it has several limitations that need to be addressed.

An important limitation is the memory allocation to the pipeline stages. In order to allow the execution of one lookup for every pipeline cycle the pipeline stages should not try to access the same memory space. Allocating single chip memories for every pipeline stage is a natural solution in providing zero memory contention. However, when the lookup chip is fabricated one must know how big the memories allocated to each pipeline stage should be. Compounded with this fact, is the challenge of being able to accommodate this solution on a single-chip implementation that can justify the low-cost and low-power of an algorithmic solution.

Basu et al. [2] is one of several papers in this field that identifies memory balance as a critical issue in the design of IP lookup engines. Their technique to reduce memory imbalance is to design the tree structure to minimize the stage that has the largest memory. However, assuming a fixed size trie and an 8 stage pipeline for IPv4 their results show that for different IP databases, the memory in some stages still varies dramatically both from stage to stage in the same application, as well as between same level stages in different applications. Even with their new algorithm, the memory allocated to one stage varies from nearly 0 to 150Kbytes for various IP tables (of sizes between 100,000 and 130,000 prefixes). The worst case bound for a million prefixes is 11 Mbytes per stage or 88 Mbytes across all eight stages. Sizing the ASIC for the worst-case memory bound would be extremely expensive. In some cases, the total allocated memory can be an order of magnitude bigger than actually needed.

To address this imbalance, one can use complex dynamic memory allocation schemes (which dramatically increase the hardware complexity) or over provision each of the pipeline stages (which results in memory waste). The use of large, poorly utilized memory modules also results in high memory latencies, which can have a detrimental effect on the speed of each stage of the pipelined computation, and thus on the throughput of the entire architecture.

This is one example of the kind of complex architectural decisions that we had to address in the design of a single-chip high speed IP lookup hardware solution. Our paper describes NSE (Network Search Engine), a single-chip, high-performance, pipelined implementation of a multi-bit trie based search algorithm. The algorithm itself (not discussed here) is an improvement in storage capacity over the one designed by Eatherton et al. [3]. The implementation may be used as a high-speed IP lookup Co-processor which has the primary application of accelerating IPv4 and IPv6 prefix look-ups. It can support searches at a rate of up to 250Msps on up to 4096 routing tables and can store up to 512K 32-bit Classless Inter-Domain Routing (CIDR) IPv4 prefixes. It also provides fast update operations (insert and delete) at a rate of 250K routes per second using an embedded CPU core.

Deleted: needed to be

Deleted: ed

Deleted: actual

Deleted: w

The rest of the paper is organized as follows. Section 2 introduces the IP lookup problem and describes the compressed multi-bit trie algorithm used in the NSE. In Section 3 the architecture and micro-architecture features of the hardware implementation of the algorithm are described. Section 4 introduces the role of the firmware and the embedded CPU in the device. We share some experimental results on routing table capacity in Section 5. The design methodology, modeling tasks and flows are described in Section 6, followed by conclusions in Section 7.

2 IP Lookup Problem and Algorithm Description

Prefix	Value	Next Hop
P_1	0000001*	NH_1
P_2	0000000000*	NH_2
P_3	01101100*	NH_3
P_4	0110110100*	NH_4
P_5	0110110101*	NH_5
P_6	11001*	NH_6
P_7	111101000*	NH_7
P_8	11110101*	NH_8
P_9	11110101110*	NH_9
P_{10}	01100*	NH_{10}
P_{11}	011011*	NH_{11}
P_{12}	*	NH_{12}

Table 1: A simple example of routing table with 12 prefixes

The IP lookup operation requires a longest matching prefix computation at wire speeds. In IPv4 domain, for example, each 32 bit IP destination address of the received packet is matched using a database of IP prefixes. Each prefix entry consists of a prefix and a next hop value.

For a better understanding of the problem let's consider the following simple example based on an IP lookup database consisting of the following 12 prefixes shown in Table 1. If the router receives a packet with the destination address that starts with *11110101110* then the next hop value associated with the prefix P_9 is selected.

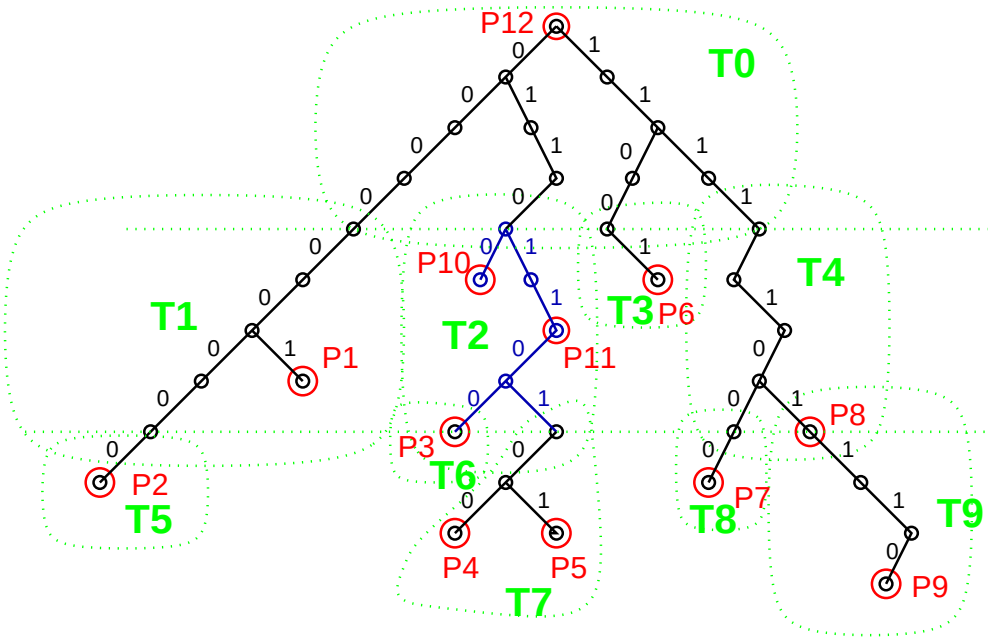

Figure 1: The trie lookup structure associated with the routing table in Table 1.

2.1 IP Lookup using Multi-bit Trie

There are many solutions in the literature for the IP lookup problem ranging from binary search to trie lookup[4]. The most common data structure for doing prefix lookups is some form of *trie*, where a trie is a tree where branching decisions are made based on values of successive bits in the destination address. Figure 1 shows a trie lookup structure associated with the routing table given in Table 1.

In the case of IPv4 lookup operations using 1-bit tries the overall latency is given by the height of the trie which equals the maximum number of bits in an IPv4 prefix, which are 32. In order to reduce the overall latency one can use *multi-bit tries* instead. A node in the multi-bit trie is associated with a subtrie in the original 1-bit trie. The number of bits that are used to identify the next child node to be traversed in the multi-bit trie is called a *stride*. A trie with all its strides having the same width is called a *uniform stride trie*. If the stride size is 4 then the overall latency of an IPv4 lookup reduces to 8. The dotted lines in Figure 1 mark the subtries with fixed depth(for example 4) in the original 1-bit trie.

Deleted: is

The multi-bit trie groups all branches in the same level with the same parent stride value into a table. This is called a *trie table*. If the prefix is divided into an array of n-bit strides, the maximum possible entries in the next level table is 2^n. The algorithm encodes

all next level stride values from same parent into a 2^n-bit data field and stores it in the entry in the parent trie table, along with the base address of the next level trie table. We call this the *CBV*. The data structure storing this information is called a trie-node. Table compression is achieved by allocating memory for the actual number of tables that exist instead of the maximum size of 2^n. For the last stride of each prefix, a similar type of data structure is used, except in this case the pointer is pointing to a table containing next hop information, called the *nexthop table*. We call this entry the *PBV*.

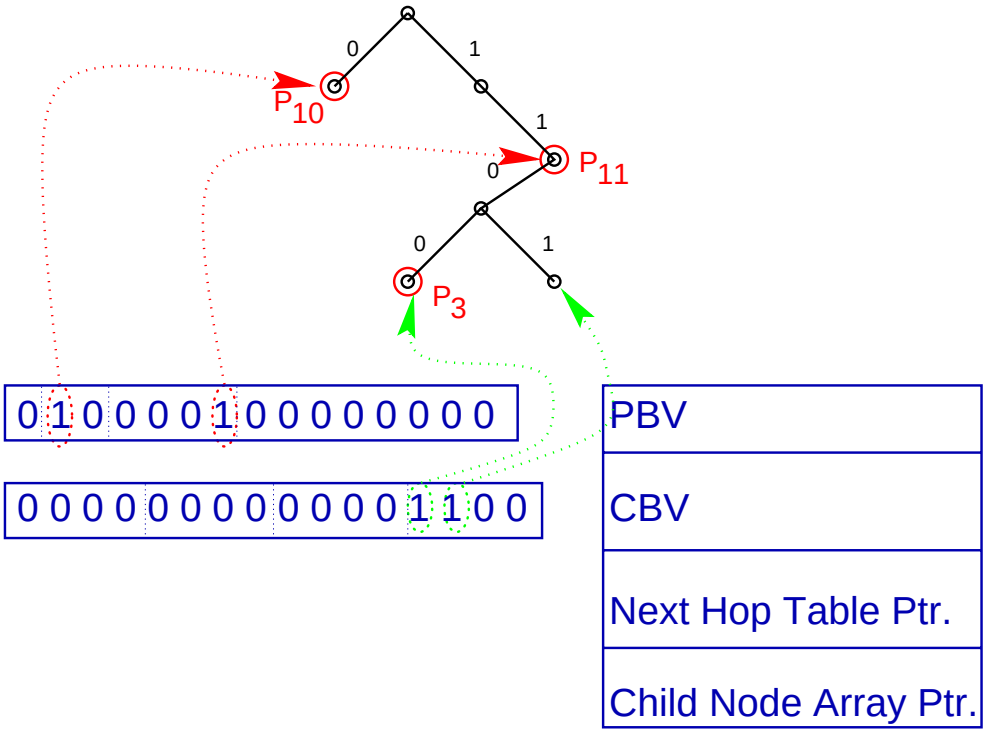

Figure 2: Each subtrie in the original trie is represented by a node in the multi-bit trie. Each node has associated two tables that represent the child nodes as well as next hop information associated with prefixes that are located in the current node(subtrie). This picture shows the representation of the subtrie T_2 from Figure 1.

Figure 2 shows the representation of the subtrie T_2 from Figure 1 and two tables that store the child nodes and the next hop information associated with the current node.

The first bit vector *PBV* describes the distribution of the nodes associated with valid prefixes inside of the subtrie. Two bits are set in *PBV*; they correspond to the valid prefixes P_{10} and P_{11} existent in the subtrie as it is shown in Figure 2 [1]. This bit vector represents a linearized format of the original subtree: each row of the subtree is captured

[1] The node associated with the prefix P_3 does not belong in this subtrie. Instead it is the root node of one of its child subtrie.

top-down from left to right. Each bit is associated in order with the prefixes: *, *0**, *1**, *00**, *01**, *10**, *11**, … , *111**. The next hop information associated with all the valid prefixes in a subtrie may be grouped together in the same area for which the data structure may need to keep only a base pointer and use an offset to find the right data.

The second bit vector *CBV* describes the child distribution. There are at most $2^4=16$ children and a bit is set whenever a child exist. Otherwise, the bit is 0. Thus, in Figure 2 we only have two bits set corresponding to two child sub-tries associated with the prefixes *1100* and *1101*, respectively.

This representation of the multi-bit trie node contributes to a reduction in the memory space occupied by the search structure. However, this node format does not take into account different shapes the sub-tries may take. For example it is very common for a sub-trie to contain a single path that may have a different compressed representation. In order to accommodate different trie shapes the NSE algorithm allows for different node types that are not detailed in this paper. These node compression techniques allow us to reduce the overall memory size of the search structures on average by 30% comparing with other state-of-the-art techniques [3].

2.2 Executing Searches in a Forwarding Engine

A search operation executes as follows. Assume that we need to identify the longest matching prefix associated with a destination address 01101101010. The algorithm considers strides of 4 bits of address at a time. It starts by reading the child bit vector associated with the root node and it determines if there is a child sub-trie with the root at the position 0110. This corresponds to the seventh bit in the CBV being set. This bit is set which means that the search continues to the next node by using the next four bits of the address. In parallel it determines if there is any matching prefix in this node. If there is a match, the algorithm remembers it and continues the search recursively by going to the next child node. When the search fails, the last matching prefix represents the longest matching prefix for the search.

To achieve high throughput of 250Msps, the tree traversal can be pipelined across a number of stages. This requires allocating memory to each pipeline stage to store state used in each stage's computation. For example, to pipeline a trie, the stage I memory may store the trie nodes at level I. Prior work has shown that such tree allocation to pipeline stages is unevenly distributed, because the trees are typically unbalanced. For example, binary tries on typical IP prefix tables are highly unbalanced. As a result, despite a wide variety of academic and commercial solutions, only a few solutions do well in terms of performance, efficiency, and cost.

Static allocation of memory at each stage results in memory waste. Complex dynamic memory allocation schemes increase the complexity, and directly affect the latency of the task at each stage. In the next section we describe the NSE architecture that uses a dynamic scheme to allocate on demand fixed size memory blocks to the pipeline stages. The scheme uses a configurable cross-bar to switch between the prefix search processing

Deleted: bank

blocks and a set of memory blocks. It allows the pipeline to execute up to 16 concurrent accesses each cycle to different memory blocks.

3 Network Search Engine Micro-architecture

The pipeline architecture for the Network Search Engine implements the multi-bit trie implementation previously presented, though it is not limited to this solution. The architecture supports:

- Longest Prefix Match of up to 512K IPv4 routing table entries.
- Support for 32 and 64-bit searches (IPv4 and IPv6).
- High search throughput at OC-768 rates, viz. 200-250M searches/sec
- Compatibility with LA-1, interface proposed by the Network Processor Forum (NPF)
- Support for transparent table maintenance of routing table and next-hop table entries
- Acceptable prefix update rate.
- Support for logically independent tables, to enable VPN and MPLS features.

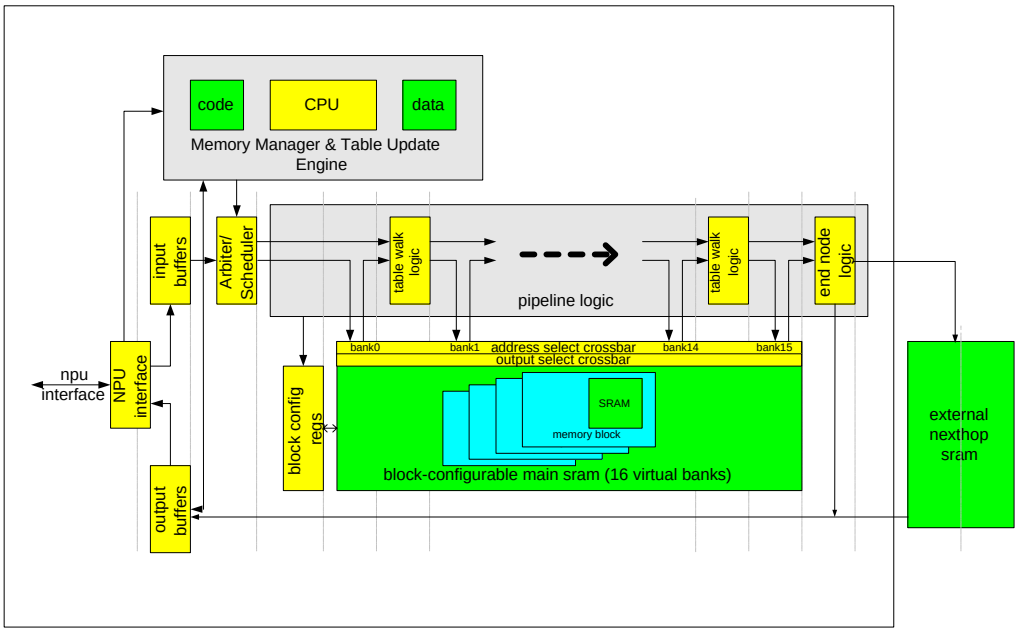

Figure 3: The Network Search Engine Micro-Architecture

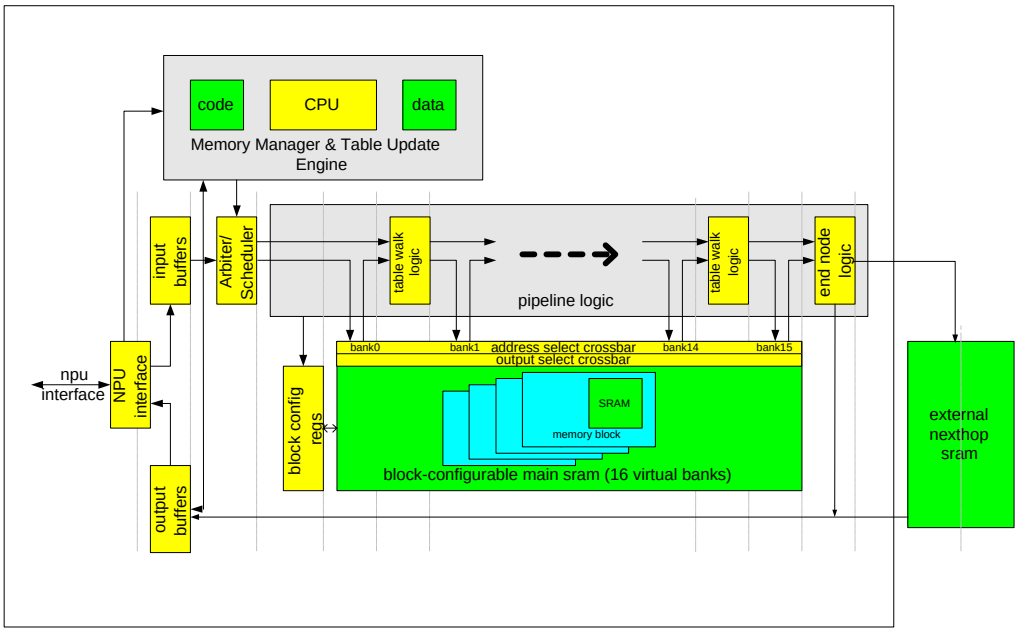

Figure 3 shows the Network Search Engine micro-architecture. An on-chip SRAM is used to store the IP prefixes. It contains the data structure required to implement the algorithm. This includes a representation of the multi-bit trie table with pointers connecting the successive levels of the trie. The advanced optimization techniques of prefix packing and prefix compression help us achieve extremely high storage efficiency of prefixes. The external next-hop information associated with a given prefix is stored in off-chip commodity SRAMs. The on-chip tables only contain a pointer to the next-hop tables. When configured in an index-only mode, next-hop indexes are permitted to reside on-chip, at the cost of reduced routing capacity.

In order to achieve throughputs of the order of 250MSPS (searches per second), the tree traversal is pipelined across a number of pipeline stages. The search pipeline contains 16 banks, each of which can process 4-bit prefix segments, enabling support for 64-bit IPv6 searches. In each bank, the processing of the 4-bit prefix segment requires access to the nodes in the multi-bit trie at that level. So each bank needs memory from the on-chip SRAM to store the nodes associated with that level in the trie. Since the shape of the trie is very dependent on the distribution of prefixes in the routing table, the memory requirements at each bank can be unevenly distributed.

Our architecture uses a dynamically configurable scheme that can allocate on demand fixed size memory blocks to the banks. A fully configurable memory crossbar network is used to enables access to any one of over hundred SRAM blocks from any of the 16 banks. This provides for more efficient storage for the prefixes, and makes the NSE less immune to variations in routing table distributions. The memory requirement at each bank can grow dynamically to the desired size needed by the multi-bit trie representation, without over allocation and memory waste.

Comment [LBH1]: Do we need to reveal this?

Deleted: 100

While the primary task of the Network Search Engine is to respond to IP look-up requests, it also provides transparent support for updates, *i.e.* inserts and deletes of IP prefixes to the routing table. An on-chip embedded CPU is responsible for maintaining an optimal trie representation of the routing in hardware while managing insert and delete requests and managing the trie table and the next-hop table memories. All this is done without interrupting the search throughput while maintaining an acceptable prefix update rate.

Deleted: *i.e*

At a system level, the Network Search Engine functions as a co-processor in pass-through or look-aside configuration, to offload the look-up functions from the network processor (NPU). It employs two standard, low cost, simple, high-performance QDR-II interfaces. The interface to the NPU is a 32-bit slave interface, configurable to 16 or 32 bits. In 16-bit mode, the interface is fully compatible with the Network Processor Forum (LA-1) interface and capable of full-bandwidth transfers (up to 250Msps) for IPv4 prefixes. For 64-bit prefixes and 64-bit nexthop data, the 32-bit interface is used. The external (next-hop) SRAM interface is also a 32-bit QDRII interface that matches the NPU interface in width and frequency. It supports various configurations of QDRII SRAMS to provide 256K or 512K 32-bit or 64-bit next hop data.

Deleted: upto

3.1 Search Pipeline Implementation

The search pipeline implements our multi-bit trie search algorithm. In each bank of the pipeline, a 4-bit prefix segment is processed. The logic in each bank receives two entries from the memory blocks of the previous bank. These entries represent two nodes from the trie table at the previous level. The logic at each bank is responsible for determining the address of the next level (if traversal must continue) or the address of the next-hop table (if an end node is reached). Up to 16 possible node types can be encountered. These fall into 2 major types - a *child or trie node* and an *end-node*.

The child node contains the pointer to the table at the next level and a run-length encoded index for each entry. The end-node contains a pointer to the next-hop table and a run-length encoded index into the next-hop table. For child nodes, the bank search logic uses 4-bits of the prefix segment (associated with the level), to position index into a bitmap of the child node, determine if there is a match for the prefix segment and then calculate the exact address of the next-level trie table. End-nodes can be of many types. In its simplest form the bank search logically positions indexes into the end-node bit map to determine a 0,1,2 or 3-bit prefix segment match and accordingly calculates the exact address of the next-hop data associated with the matched prefix ending at this level. Other end-node types perform various forms of prefix packing and prefix-compression. In this case the bank search logic matches longer prefix segments (up to 16-bits) and calculates next-hop addresses for matched entries.

The search pipeline handles exact and longest prefix match searches. In the latter case, encountering a successful match does not end the search. Instead the next-hop address is saved and staged forward until the search finally fails. The last saved next-hop address is associated with the longest matching prefix. Once a child-node address for the next-level is computed, this is sent to the memory blocks associated with this level. A look-up is performed and the data from memory is then sent to the next level. In clock

cycles, each bank of the search pipeline lasts 4 clocks, with the logic and memory lookup lasting one clock each and the other 2 clocks devoted to interconnect. In addition to its primary function, the search pipeline also provides support for various other commands issued by the embedded CPU, during the execution of prefix update operations.

3.2 Memory and Crossbar Implementation

As mentioned earlier, the SRAM for the trie memory is implemented as a flexible, virtual 16-ported memory model with a crossbar structure that permits any bank to access any one of the SRAM blocks. In any given cycle up to 16 concurrent reads to 16 different memory blocks is possible. Since writes to the trie memory SRAMs are relatively infrequent (needed only during updates and memory management) write support is provided to only 1 SRAM block in a given cycle. Each bank logic block generates 2 addresses. An address crossbar network routes the address from each of the 16 banks to any one of the memory blocks, and the data crossbar network routes two data words from up to 16 SRAM blocks to anyone of the 16 banks. Bank configuration Registers are used to identify the bank that an SRAM block belongs to. A logical view is shown in Figure 4.

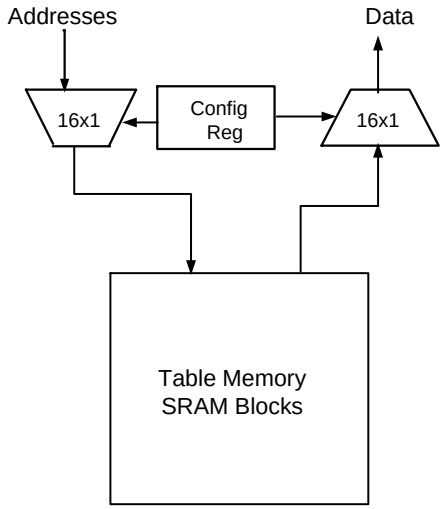

Figure 4: Logical View of the Trie Memory Crossbar Network

The basic SRAM memory block is a *limited stride dual-port memory*. It is capable of reading 2 data words every clock, which are a limited distance apart. This constraint simplifies the addressing mode. In addition, the memory size is close to that of a single-port memory, while satisfying the multi-bit trie lookup algorithm requirements of 2 entries from each trie table at every level.

Deleted: , result

Deleted: ing in a

Deleted: that

For each one of the 16 banks, the task of presenting an address, accessing the memory and reading data out or writing data in to the memory lasts 3 cycles.

- In the first cycle (A) the generated address is routed across the address crossbar to select the desired SRAM block, while disabling all the other blocks.
- In the second cycle (M), the selected SRAM is read with the presented addresses and 2 words are read at the output. If a write needs to happen, then only one of the two addresses is used and data is written into the SRAM.
- The 3rd cycle (D) is again interconnect dominated; consumed in routing the result of a read across the data crossbar to the logic stage for the desired bank

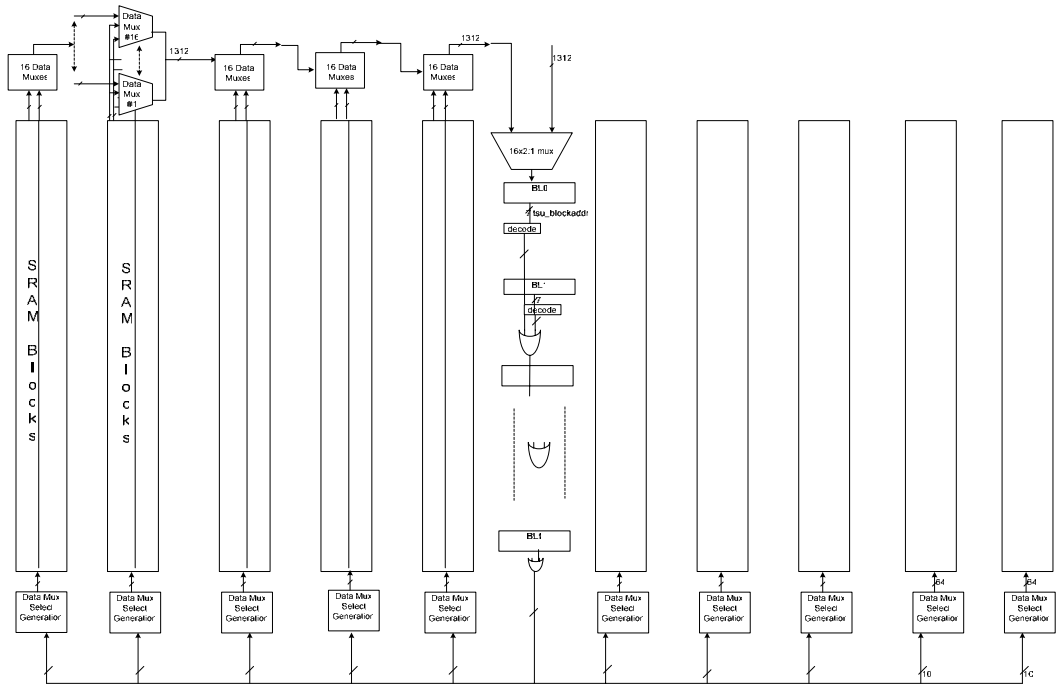

Figure 5: Global Physical View of Memory and Crossbar.

Figure 5 shows a global physical view of the trie memory SRAMs arranged in columns on either side of the search pipeline logic. As it can be seen, the time to route an address or data word between an SRAM block and its associated search logic block can depend significantly on how far apart they reside physically. Instead of constraining the cycle time to be a function of the worst-case wiring latency between a source and destination, a more global approach was adopted, employing two key strategies. First by delaying/skewing the clocks (also known as *useful skew*) to the SRAMs, time was stolen from the address path to give to the datapath or vice versa. Second a "*wave-pipeline-like*" approach to clock distribution allows clock repeater delays in one path to cancel out the delays on the other path. Consider the clock distribution scenario for the address path shown in Figure 6. Each group of 4 search logic blocks clocks their address one repeater "dly" apart. On the other side, each group of 4 SRAM blocks are clocked one "dly" (plus some useful clock skew(*ucs*)) apart. An address from the top group meant for the second

group of memories will arrive one "dly" later. But the clock to this group of memories is also delayed by one "dly" allowing the address to arrive late. So the address repeater delay is cancelled by the clock delay. The longest delay for the address is not the U shaped distance, but 2/3rds of it.

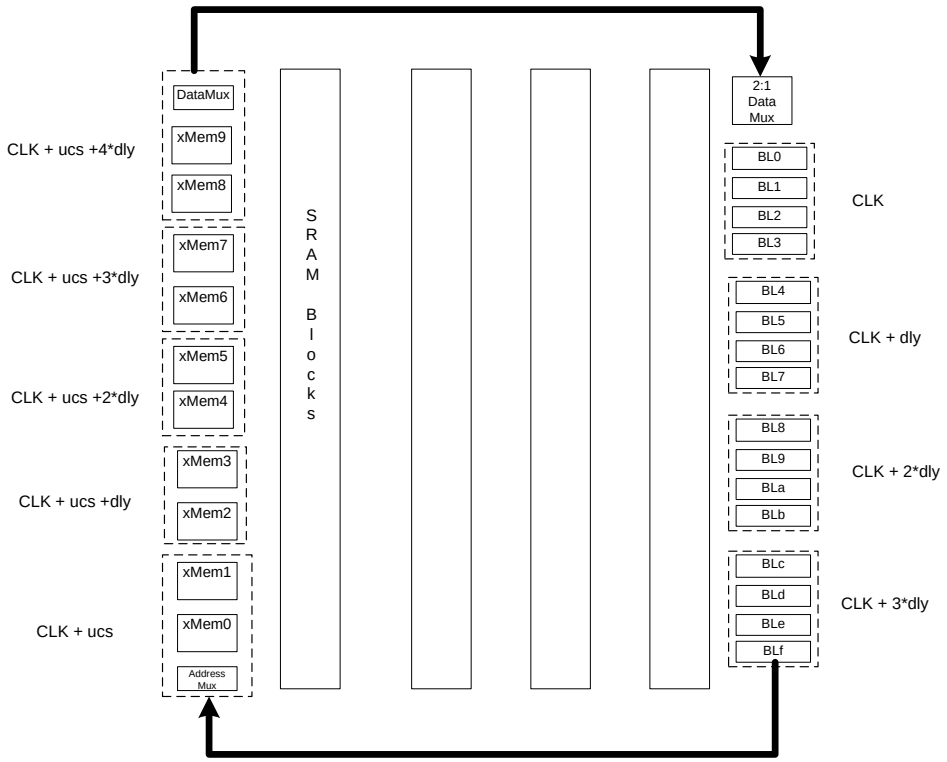

Figure 6: Clock Distribution in the Crossbar and Memory

3.3 Physical and Logical System Interfaces and Programming Model

The 250MHz 32-bit QDR-II physical interface on the NPU side is a fast, narrow interface that can achieve bandwidths of the order of 4.5GB/s. This is made possible with concurrent reads and writes, and double data rate transfers (on each edge of the clock). High-Speed Transceiver Logic (HSTL) I/Os with source synchronous clocking help compensate for output delays and flight times. The NPU interface unit of the NSE receives the data clocks (NC,NC#) (used to clock the NSE outputs) from the NPU, and uses an on-chip DLL to delay and de-skew them, before using them to clock the output(NQ) to the NPU. It also generates the echo-clocks NCQ/NCQ#, used by the NPU to capture the NSE's output data. The physical interface to the next-hop SRAM is also a QDR-II interface capable of delivering similar throughput. The difference is that, since the NSE is a master on this interface, it generates the clocks (SC/SC#) used by the

SRAMs to place the output data. Echo clocks SCQ/SCQ# from the individual SRAMs are used to capture the read data arriving from memory. A pair of DLLs (in the NSE SRAM interface unit) is used to align them at the center of the returning data. The NPU and SRAM interface source-synchronous clocks can be independent and different from the clock used by the NSE core, which in turn can be different from the embedded CPU clock; the only restriction being that the core and the SRAM interface cannot run slower than the NPU interface.

The LA-1 interface standard defines only short, fixed-delay, read and write transactions, but does not specify a logical layer. So a longer, variable delay operation like a prefix search or insert is handled as a compound operation via the logical protocol. A search command is sent to the NSE via the NPU interface write transaction, and the response is read later via an NPU interface read transaction. For maximum flexibility, the NSE also allows search status to be defined as a separate field within the result, allowing search status to be read independently of the result. Search results can be retrieved sequentially in the order in which the search commands were received or in random order with a context-id used to track the search. This enables out-of-order return of search results. Leveraging the fact that next-generation NPUs are multi-threaded, multi-context polling is supported allowing up to 128 active search requests initiated by these threads. A 128 deep search results buffer in the NSE acts as an in-order FIFO or as a random-access register file when indexed by the context identifier to randomly retrieve search results. In addition to search requests, the NSE also supports several other types of command including prefix insertions, prefix deletions, and various control and maintenance commands. However, the throughput requirements of these operations are not as demanding as that for search requests.

Deleted: by

Comment [LBH2]: First use. Use NSE

Deleted: IU a

Deleted: so

4 Managing the Search Engine

The Network Search Engine (NSE) contains three functional entities: a search engine trie-table structure using an internal SRAM(TM)[2], an embedded CPU with an associated memory and an external memory(EM). The search engine generates indexes to access the next hop information that is stored in the EM. The indexes are calculated based on information stored in the trie-entry.

The internal SRAM is made up of fixed-size blocks that hold trie node information. The external SRAM holds next hop table entries. Each SRAM block can be allocated to any of the pipeline stages. However, only one pipeline stage can access an internal SRAM block at a given moment.

Comment [LBH3]: number

Deleted: contains

Deleted: over 100 f

Deleted: memory

Trie tables and next hop tables in the internal and external memories respectively, are allocated in variable size units. The allocation policy on both memory systems is managed by the embedded CPU. The system requires the allocation of as many as possible next hop entries (500K entries) with high memory utilization without affecting the search capabilities of the circuit.

Deleted: a high

Deleted: as well as

[2] All the node information is stored in SRAM.

4.1 Requirements for External Memory Updates

The Network Search Engine allocates *variable size next hop tables* in the External Memory (EM). A table may have between 1 and 16 entries. A next hop entry may have 36 … 72bits. Each child node in the trie memory has a base pointer to the associated next hop table. A next hop table can be referenced by at most one parent node.

In the ideal case we would like to be able to fit 500K next hop entries in 500K words, where a word is of the size of an entry. This can be easily done if fast updates are not part of the requirements. However, the dynamic behavior of the NSE which must allow the insertion/removal of entries concurrently with the searches without affecting the correctness of the results, may contribute to fragmentation in the EM allocation.

A standard result in the memory allocation literature states that no allocator without doing memory compaction can have a utilization ratio better than $1/\log_2 W$ where W is the largest possible allocation request(in our case W=16). Since this is unacceptable, we consider memory management schemes with compaction. Compaction refers to moving allocated tables (blocks) to increase the size of the holes in between.

However, compaction has several problems. First, moving a table M in EM requires correcting the pointer in the parent that points to M. This may be simply done if the memory manager keeps track for each table where its parent is located. It means that for each table the memory manager may need to keep a pointer to the parent. A second problem is that the literature talks mostly about global memory management. However, in NSE we address the problem by doing local memory compaction trying to get a high compaction ratio.

Deleted: simple to be done

4.2 External Memory Management

We now present the design of the External Memory(EM). As mentioned before it is intended to be used only for storing next hop prefix information without any additional data. This implies that the *state* information regarding the memory status as well as the parent pointers, if it is the case, are stored at the embedded CPU level.

We first reduce the complexity of our design by organizing the memory into *segments*. Each segment has the size of *64 entries*. For each segment the CPU stores only the location of the first available entry in the segment. Also, each segment is associated with only one internal (trie) memory block. The embedded CPU keeps a list of associated segments in EM for each internal memory block. As a result the parents of the tables allocated in the current segment are all residing in the same memory block. The memory tables are allocated contiguously inside a segment. This means that the memory management algorithm tries to guarantee that at any moment in time there is no hole in between two blocks allocated inside of the same segment.

Deleted: of

In the case of a delete operation the memory manager may need to fill up a possible space created in between two tables allocated to the same segment. In order to

do so, it needs to identify: (1) all the tables that are allocated in the segment, (2) the parents of the allocated tables. Both operations are initiated by the embedded CPU and use *hardware-assisted search* features implemented in the pipeline stages. If table movement is required, all the remaining tables in the segment are first moved in a cache segment in the External Memory. In a second step all the parent pointers to the moved tables are updated. In the end the original segment is emptied and becomes the new cache segment for the external memory[3].

The information associated with the segment occupies 64Kbits in the CPU memory (for an external SRAM capable of accommodating 512K entries), and increases to 128Kbits (for 1024K next hop entries). The information to represent the association between each bank and the segments in the EM uses about 160Kbits of the embedded CPU's memory

4.3 Trie Memory Management

The Network Search Engine uses a search structure made up of a set of trie-table structures which are stored using an internal SRAM that we call Trie Memory(TM). This trie memory consists of over 100 fixed-size SRAM blocks. Each block can accommodate multiple trie tables, and each of the trie tables may have up to 18 entries.

Comment [LBH4]: number

Deleted: about

The NSE uses TM in conjunction with a pipeline consisting of 16 banks of search logic, each lasting 4 stages. This introduces a limitation in the way entries are allocated: there can not be two entries in the same memory block that can be accessed from two different banks, or in other words a memory block can not be accessed from two different banks. As a result, the CPU keeps a *bit-vector* for each bank, in which each bit i is set if and only if the memory block is used in that bank. A 17th bit-vector (*OR bit-vector)* is kept as an OR result of the 16 other bit vectors.

From a memory manager perspective, the TM requirements are similar with the requirements for the memory manager for the External Memory with the slight modification that we mentioned above.

Compared to the previous case in which the system itself does not impose constraints regarding where the parent nodes are located, in this case a trie table stored in a memory block associated with a bank i, have its parent stored in bank i-1 of the pipeline. This information may reduce the number of bits of data one may need to store in order to identify the parent node[4].

Deleted: has

4.3.1 Trie Memory Allocation

[3] All these steps are required because the NSE keeps the search structures consistent. The reader should recall that in NSE several searches can execute while an update task takes place. However, there can be only one update task in the execution stage.

[4] As we mentioned before a quick identification of the location of the parent node table is crucial for the efficiency of any compaction algorithm we may use.

Trie Memory Block Allocation: A TM block needs to be allocated whenever, during an update operation, there is no more space left (for insertion) in all the memory blocks that associated with the current bank.

The first available block is identified by the position of the first bit that is zero in the OR bit vector.

Entry allocation in the Trie Memory: Similar to the next-hop table entry allocation in the external memory, the base unit for allocation is a trie-table, which may have up to 18 entries in TM. The insertion of a new route in the NSE may either increase the size of an already existing table by one, or result in the creation of a new table. In the case of a route removal it contributes either to a decrease by one of the number of entries in the original table or to the disappearance of the whole table.

There are two major problems to be addressed: (1) the allocation of the entries inside of a memory block, and (2) the identification of the parent of a trie-table.

A solution to the first problem uses two bit vectors (ALLOC and START), for each memory block. Each bit-vector has as many bits as there are entries in the memory block. A bit in ALLOC is set to show that an entry is allocated. A bit in START is set only if it corresponds to an entry which is the first entry in a table. The embedded CPU uses about 1Mb of data to store these bit vectors.

The second problem may be solved by storing parent information with each of the tables. In this case the entries are organized in segments of 64 entries. For each segment, the CPU keeps a list of tables. Each table is able to host up to 7 parent pointers. The memory space that is occupied in the worst case is at most 1.4Mbits considering that there is an average of about 8 entries per trie table.

Comment [LBH5]: is this correct? The whole CPU has only 2Mbits of memory

Deleted: it

Deleted: is

4.4 Evaluation of the Memory Management Designs

Without knowing the information regarding the location of the parent nodes the memory manager needs to execute an exhaustive search through the memory block where the parent may be located. The cost of this *parent search* masks the cost of any other steps that may occur into the system performing an update. This cost is a function of two variables: the number of entries in a trie memory block and the availability of spare non-search cycles in the search pipeline to use for an *update* operation. Since the block sizes are fixed, the update rate is largely dictated by unused search cycles. In a situation where a search enters the pipeline every other cycle, a parent search takes about 9000 cycles.

Deleted: ban

Deleted: search

Comment [LBH6]: this reveals the block size!

Deleted: For example, let's consider the system working on a clock at 250MHz with a data flow with a rate of 8ns. In this situation the Memory Manager may use any other cycle to execute operations that are related to the update. Therefore an exhaustive search in a trie memory bank with 4K entries will take on the order of 9000 cycles.

Significant differences between schemes occur only if parent pointers for each of the next hop tables are known by the Memory Manager. In this case the time for an update is determined mostly by the read/write operations in the External Memory. In the worst case scenario 64 entries are read and 64 entries are written in the EM. In this case the update time does not exceed 300 cycles.

Deleted: exceeds

Deleted: about

5 Routing Table Capacity

We made a step further. The multi-bit trie algorithm in [3] does not take into account the shapes of different subtries that may exist. As a result, we have noticed that there is space for improving the amount of memory used by the search structures. We developed a new multi-bit trie algorithm that tries to take into account all this variations with only an insignificant increase in the computation complexity associated with each node.

Deleted: ’

We investigated the performance of our algorithm reflected in the overall memory space that is used by the search structures. In our evaluation we used both real life routing tables as well as synthetic generated ones.

The real life routing tables were extracted using instances of the BGP routing tables available at RIPE[5] and RIR[6] on Sept. 22, 2003 and parsed using the software available at [7](route_btoa). We extracted the routing tables associated with ATT(AS number 7018), Sprint(AS number 1239), Level 3 Communications(AS number 3356) and France Telecom(AS number 5511). In order to further test the algorithm with larger size routing tables we used synthetic generated tables based on two different models of routing table growth: a model developed by the Network Processing Forum (NPF)[8], and a model developed by Narayan et al[9]. The synthetic generated tables reflect what both the industry community as well as academic community believes to be typical routing tables.

Unlike TCAMs which have fixed prefix storage capacity, the actual storage capacity of the NSE is sensitive to the dataset. Also as a pipelined device that uses dynamic memory allocation NSE is also sensitive to the memory allocation routines.

Memory Management Overheads: Our search engine employs segment-based memory allocation schemes. The memory can be grouped into internal memory that stores the CBV tables that contain information related to the child nodes and the external memory that stores the PBV table that contain information related to the prefix nodes and implicitly to the associated next hop data.
The internal memory is organized into fixed size banks. Each bank is made up of a fixed number of segments. Any memory bank can be allocated to at most one pipeline stage. The external memory is organized into fixed size segments. Our simulation results show that this fragmentation introduces in practice an overhead of 1.5% in the case of the internal memory and at 0.86% in the case of external memory.

Network Search Engine sensitivity to the data set: We further investigate using a set of sample synthetic routing tables with increasing sizes. The table size changes from 50K prefixes to 500K prefixes in increments of 50K. In both cases of tables generated using the NPF model[8] as well as Narayan's model[9] the internal memory utilization and external memory utilization is linear in the number of prefixes. Each 1000 prefixes uses 0.14% of the internal memory and 0.20% of the external memory. This counts for about 70% of the overall internal memory space. In the case of the real routing tables, 1000 prefixes use 0.13% of the internal memory space and 0.20% of the external memory space. These results include the overhead of the dynamic memory management process.

6 NSE Design Methodology

Since the design of the NSE started all the way from analyzing storage-efficient algorithms for the IP lookup problem to the design of an SoC with an embedded CPU core, several different models were used during the course of the design, from algorithm to hardware.

At the algorithm and architecture level, functional models were used to validate the multibit trie algorithms and the partitioning of the software and hardware components of the NSE device. The models were basically written in C++, with SystemC interface wrappers added during integration. Two versions were developed and maintained for different purposes. The first version, called **SDM**, was developed largely as an algorithmic model with some consideration of hardware structures and partitions, especially in the model of the main search pipeline; so some implicit hardware/software partitioning was already done. This version was mainly used to validate functionality and the storage capacity of the various optimization schemes used in the early stage of the project. We later found a way to extend its use to facilitate RTL validation. The second version, called **HWM**, was based on the hardware portion of the SDM. It interfaced with the firmware running on a CPU. The role of the firmware was to manage the updates, and the trie tables and next hop tables in internal and external memory. The firmware that the HWM interfaced with ran on the native machine or the ARM ISS simulator. The HWM model enabled the quick development, testing and verification of the NSE firmware.

Once the hardware-software partitioning was complete, we used a behavioral model to develop an executable specification of the micro-architecture. This model, called Hardware Development Model **(HDM)**, was implemented in SystemC 2.0, with gnu tools running on Red Hat Linux. We felt this was a very effective low-cost approach to model development. The HDM was intended to serve as documentation, as well as a reference model for the implementation (RTL) team. It was also intended for firmware development, but had to be abandoned in favor of the HWM as the project progressed. The primary reasons were: lack of maturity in the HDM when firmware needed it, and the slow run times in the HDM, which made it impractical to run effective simulation cycles for firmware development. The HDM was highly detailed in specifying the logic partitioning, block definition and interfaces. This enabled the implementation team to work more or less independently on the RTL. The HDM was also useful for verification tasks. Since the HDM and the RTL matched sub-block interfaces, cycle-by-cycle comparisons were possible between the RTL and HDM. With cycle-by-cycle comparison with the HDM all possible node formats and modes, including early search retirement in the search pipeline was tested effectively.

The RTL was implemented at a different location from where the architecture and micro-architecture was developed. The detailed block and sub-block level implementation of the HDM (and the micro-architecture) made for lesser design unknowns for the RTL. Aside from clock domain synchronization and I/O timing issues at the NPU interface, there were no major design issues encountered during RTL implementation.

Figure 7 shows the models used at the different levels and their role in firmware development and validation. As it can be seen, every phase of the firmware development process posed a different set of requirements and we were able to use the models at each level to meet those requirements.

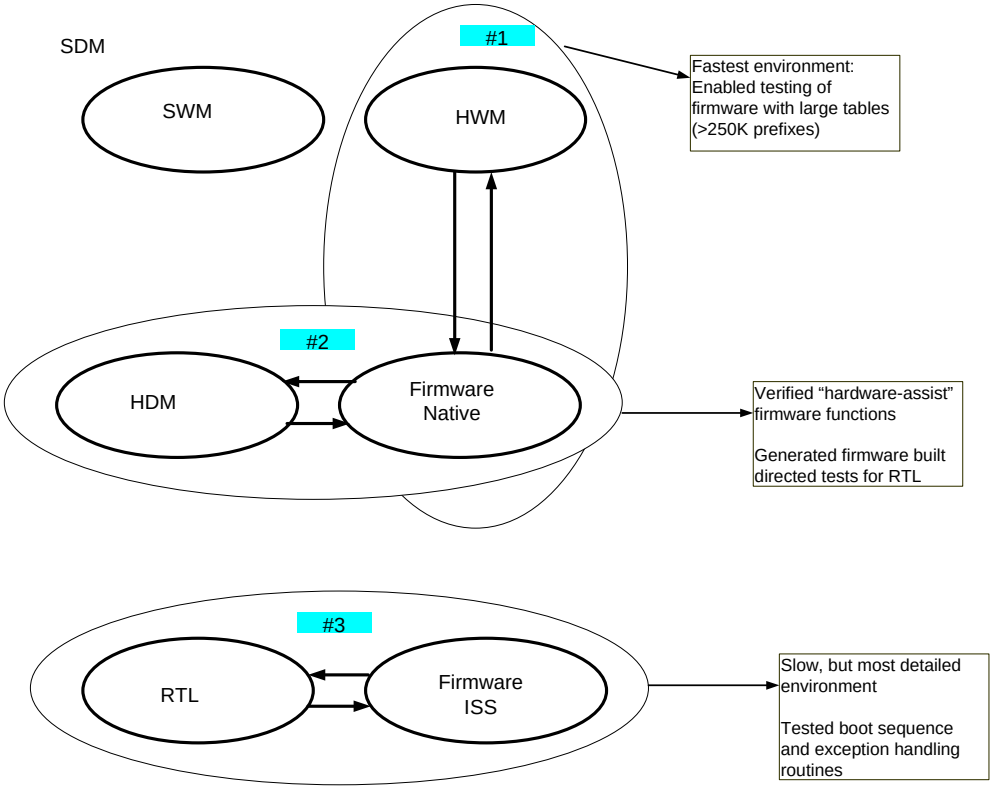

Figure 7: NSE Models and their role in firmware development

7 Conclusions

In this paper, we have presented a hardware implementation of a multi-bit trie IP lookup algorithm. We described a pipeline architecture capable of providing IP lookups for IPv4 and IPv6 packets. Our solution in conjunction with a multi-bit trie based IP lookup algorithm was implemented in a single chip SoC. It can support routing tables of up to 512K IPv4 prefixes. It provides a search rate of up to 250Msps and a configurable crossbar implementation that maximizes the amount of memory available at each bank. Interfaces to the NPU and to the external SRAM are provided through a standard 36-bit QDRII interface, which is also compatible with the 18-bit LA-1 interface described by the Network Processor Forum. The design also includes an embedded CPU to perform table management functions and implement update operations required for the routing table. By off-loading the table management functions, the NPU is freed up to do other

Formatted: Justified

Deleted: at a search throughput of up to 250Msps. The single chip SoC solution can support routing tables of up to 512Kprefixes, through

packet processing tasks. The current implementation of the NSE offers a capacity similar to that of an 18Mbit TCAM, at a cost and power that is significantly lower. The architecture is also very scalable and can accommodate 1-2M prefixes.

Deleted: version

Comment [LBH7]: Block format different

Bibliography

1. A. McAuley, P. Francis, "Fast Routing Table Lookup using CAMs", *Proc. IEEE INFOCOM 1993, Vol 3, pp 1382-1391*, San Francisco, USA.
2. Basu A. and Narlikar, G., "Fast incremental updates for pipeline forwarding engines", *Proc. INFOCOMM, 2003*.
3. Eatherton W., "Hardware based internet protocol prefix lookups", In *Washington University Electrical Engineering Department, MS thesis* (May 1999).
4. Sanchez M., Biersack E., and Dabbous W., " Survey and taxonomy of IP address lookup algorithms", In *IEEE Network Magazine, vol. 15, no. 2* (2001).
5. RRC, "Routing Information Service Raw Data", *http://data.ris.ripe.net/*, (2003).
6. David Mayer, "University of Oregon Route Views Project", *ftp://ftp.routeviews.org/pub/routeviews* (2003).
7. U. Michigan, "Multi-Threaded Routing Toolkit", *http://www.mrtd.net* (2003).
8. NPF, "Network Processor Forum Benchmark Working Group", *http://www.npforum.org/benchmarking/index.shtml"*
9. Harsha Narayan, Ramesh Govindan and George Varghese, " The Impact of Address Allocation and Routing on the Structure and Implementation of Routing Tables", *Proc of ACM SigComm 2003*, Aug 2003.