ABSTRACT

MEHROTRA, PRONITA. Memory Intensive Architectures for DSP and Data Communication. (Under the direction of Paul D. Franzon)

We focus on the design of systems where memory performance is the principal bottleneck. Two kinds of systems have been considered, an FFT engine for use in high performance DSP systems and forwarding engines used in high-speed routers.

The FFT engine was designed using Seamless High Off-Chip Connectivity (SHOCC), a high-density interconnect and packaging technology. The SHOCC substrate was analyzed for different substrate stack-ups to determine the I/O bandwidth attainable by the technology. The FFT engine was designed to make full use of this available bandwidth. It provides for rotation of data to maintain a constant stride between stages, ensuring a constant data access pattern between different stages of the FFT. Data is twiddled before storing in memory, and an efficient scheduling algorithm allows generation of twiddle factors on-chip in parallel with other operations. The memory controller uses a novel memory-mapping scheme to avoid precharge and refresh penalties in DRAMs and achieves SRAM-like access speeds.

Forwarding engines in IP routers need to perform a longest-matching-prefix search on the routing database. Two types of forwarding schemes are presented, a hardware-implementable trie based scheme and software implementations of modified binary searches. The hardware trie-based scheme uses a small amount of on-chip SRAM along with an off-chip DRAM (which stores the complete forwarding table). Only a single DRAM access is required to determine the next hop address. The binary search schemes store an additional field in the forwarding database to avoid any backtracking while searching for prefixes.

MEMORY INTENSIVE ARCHITECTURES FOR DSP AND DATA COMMUNICATION

by

Pronita Mehrotra

A Dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Department of Electrical and Computer Engineering
Raleigh, 2002

Approved by:

Prof. Paul D. Franzon
Chair of Advisory Committee

Prof. Tom Conte
Member of Advisory Committee

Prof. Gregory Byrd
Member of Advisory Committee

Prof. Douglas R. Reeves
Member of Advisory Committee

BIOGRAPHY

Pronita Mehrotra completed her 5-year integrated Master of Technology (M.Tech) at the Indian Institute of Technology, Bombay, India in 1997. She joined the Electrical and Computer Engineering department at North Carolina State University in August 1997 as a research assistant to work toward her Ph.D. Her research focus was on hardware and ASIC design. Since February 2002, she has been working as a Network Hardware Engineer at MCNC in the Research Triangle Park. Currently, at MCNC she is working on designing network adapter cards for Optical Burst Switched Networks.

ACKNOWLEDGMENTS

It has been a fun and rewarding experience working on this dissertation, and I would like to thank some of the people who made it so. My advisor, Prof. Paul Franzon, introduced me to this area and also provided much helpful guidance along the way. Paul has created a wonderful and stimulating research environment within his group, and it amazes me that he is so accessible to his students despite his busy schedule.

Prof. Douglas Reeves and Prof. Tom Conte served on my committee and helped guide my research with many valuable suggestions. I am grateful for their encouragement throughout the process. Prof. Gregory Byrd agreed to serve on my committee at the very last minute, and gave many helpful comments. I would also like to acknowledge Prof. Wentai Liu for initially serving on my committee.

Michele Joyner has been a big help throughout and she has been especially patient with my many requests as I was finishing up this dissertation. It was fun working at ERL and I would also like to thank all my colleagues at ERL who made this an enjoyable experience.

Finally, Vijay, my husband, was very patient and encouraging throughout my years at NC State. Thank you for always being there to lift my spirits whenever things looked tough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Overview of the Dissertation

Various high performance, high-speed systems are limited by the performance of memory. While processor speeds have scaled as predicted by Moore's Law, increase in memory speeds has been much slower. Even though microprocessor speeds have increased by 50x, bus frequencies have increased only by 10x [3]. This mismatch has become a critical factor in high speed, high performance systems. This dissertation discusses two such systems from DSP and data communications. The first of these is the design of a million point FFT engine and the second is the design of forwarding schemes used in IP routers. The main focus in this work is to re-engineer the memory-intensive functions of these systems in order to improve overall system performance.

1.1.1 Problem Definition: FFT Engine

High performance DSP applications, like Synthetic Aperture Radars (SARs), require extremely large computation rates and have large working data sets. As the demand for higher resolution increases, this computation rate is expected to reach TFLOP rates. These

applications involve manipulations of large volumes of data (1GB or more). Using DRAMs instead of SRAMs offers significant savings in cost, but raises additional problems due to their refresh and row access cycles. In addition, signal processing algorithms are mostly memory starved and one expects that increasing the memory bandwidth would improve performance. However, some algorithms may also need to be modified to use the increased bandwidth efficiently. Therefore, the main challenges in the design of such systems are summarized below:

1. *Achieving the maximum possible memory bandwidth.* Increasing the bus width or the number of memory channels is possible, though it is limited by signal integrity issues like noise. Thus, the underlying technology limits the available bandwidth.

2. *Algorithmic changes to exploit the available bandwidth*. Theoretically, in applications like the FFT, which have a deterministic data access pattern, it should always be possible to use all of the available bandwidth. However, this can come at the cost of additional hardware. Algorithmic changes can help to minimize this cost while providing the required bandwidth.

3. *Avoiding wastage in bandwidth associated with DRAMs*. In applications that require the use of DRAMs, DRAM precharge and refresh cycles make it difficult to fully achieve the available bandwidth.

   The FFT engine described in this dissertation addresses these issues in order to provide a high system throughput. By using a high density packaging technology, a high raw I/O bandwidth is obtained. An efficient memory mapping scheme allows full utilization of all memory channels throughout the FFT process and avoids any DRAM related overheads. Twiddling for the FFTs is done in the data path and the twiddle factors are generated on-chip to avoid storing large number of twiddle factors in memory.

1.1.2 Problem Definition: Forwarding Engine

The most time critical part in IP packet forwarding is the route lookup to determine the next hop address of the packet. As the size of the routing tables and the traffic in the Internet increase, this is becoming a very challenging problem. When an IP router receives a packet on one of its input ports, it decides, depending on the destination address of the packet, which output port the packet should be forwarded to. To make this decision, it has to look into a large database of destination networks and hosts. However, a lookup database which stored information for all possible destination addresses would be huge. For instance, for 32-bit IPv4 addresses, 4GB of memory would be required (assuming output ports are stored in 8 bit fields). For the 128-bit addressing of IPv6, an impossibly large amount of memory would be required to store all the information. To circumvent this problem, IP addresses are aggregated into blocks to reduce the number of routing table entries. Under the Classless Interdomain Routing (CIDR) scheme, routing tables only store an address prefix which represents a group of addresses that can be reached from the output port. For instance, a routing table can have an entry of 128.* with an associated output port of 2, which would mean that any destination address that begins with 128 has be directed to output port 2. There could also be another entry of 128.14.* with an output port of 5. The rules now have to be interpreted as follows: if there is a destination address that begins with 128 it should be sent to output port 2, unless it begins with 128.14, in which case it should go to output port 5. The problem then reduces to finding the *longest matching prefix* to determine the output port or the next hop address. As an example, consider a sample routing table with prefixes as shown in Table 1.1. For an incoming packet with a destination address beginning with 1011010*, entries 10*, 1011*, 101101* all give a match. The longest of these is 101101* and so the correct next hop address in this case should be 2.

Storing prefixes in the routing tables makes the tables smaller, but it also makes

Table 1.1: A sample routing table with prefixes and next hops

Prefix	Next Hop
10*	3
1011*	9
011*	1
010110*	5
001*	4
101101*	2
011010*	6
011100*	1
10111*	8
00101*	7

searching much more difficult. This dissertation looks at two different types of forwarding schemes. The first one is based on a trie-based scheme which is more suitable for smaller address sizes (as in IPv4). It is suitable for a hardware implementation and is scalable to large routing databases. The throughput is constant (does not depend on the size of the routing table) and is limited by the single off-chip DRAM access required. The other schemes are based on binary searching through the routing tables and the search time depends only on the size of the routing table. The size of the address does not matter in that case.

1.2 Novel Claims

Various innovative ideas were used in the design of the two systems. In the design of the FFT engine, the SHOCC technology was analyzed in terms of its noise and timing performance in order to exploit its maximum available bandwidth. The biggest concern in this design was the noise budget due to the 128 separate memory channels. A crosstalk analysis for the SHOCC transmission line was performed for various substrates and trace widths/pitches to arrive at a solution that could provide the required I/O bandwidth. The architectural design was matched to exploit this bandwidth. A memory mapping scheme

(specific to the FFT algorithm) was developed to avoid any DRAM refresh and precharge overheads, achieving an SRAM like performance for the overall system. Another significant design issue was the handling of the million twiddle factors (for a million-point FFT). Storing them in DRAMs would double the memory requirement and lead to poor performance. A novel twiddling scheme was suggested where the twiddle factors for the FFT are generated on-the-fly. Only a basis set of twiddle factors are stored in memory and the rest are generated from these. Scheduling the twiddle factor generation is done in cycles when the required hardware is not in use. Therefore, no additional hardware is required for this.

In the design of the forwarding engines, two types of forwarding schemes were suggested. In the first scheme, which uses a trie-based approach, a novel method to compress the forwarding table information was used to reduce the trie path-information. The compacted information can be stored in an on-chip SRAM and the final next-hop addresses are stored in an off-chip DRAM. To perform a route lookup, trie traversal is done in the SRAM and a final DRAM access is required to determine the next hop address. The amount of compaction achieved is much higher than other existing schemes making a hardware implementation feasible. In the binary search schemes, an additional field in the forwarding database is added which helps to determine the direction of search unambiguously. This avoids any backtracking that would have to be done otherwise. The binary schemes suggested have a superior performance as compared to the only other binary scheme in literature in terms of memory consumption, build time of the data structure and the search time for a given address.

1.3 Overview of the Following Chapters

The rest of the dissertation discusses the details of the two systems outlined previously. Chapter 2 reviews prior work in the design of DSP systems using high density packaging

and the various forms of FFT implementations in existence. We focus on memory addressing schemes and twiddle factor generation schemes, which were our main focus in the FFT implementation. Chapter 3 discusses the details of the physical implementation of the FFT engine. The key aspects of the design are the memory mapping scheme which enables fast, uninterrupted data accesses from DRAM and the twiddle factor generation scheme, which allows for on-the-fly twiddle factor generation. In Chapter 4 the SHOCC technology is analyzed in terms of its noise and timing performance which determines the total bandwidth available from the technology. Chapter 5 reviews existing route lookup schemes in literature. Chapter 6 describes in detail the various forwarding schemes proposed and Chapter 7 discusses and analyzes the performance of the proposed schemes. Finally, in Chapter 8 the work presented in this dissertation is summarized and some ideas for future work are discussed.

CHAPTER 2

Overview of SHOCC technology and FFT systems

2.1 Seamless High Off-Chip Connectivity (SHOCC) Technology

Seamless High Off-Chip Connectivity (SHOCC) is a combined packaging, interconnect and IC design technology aimed at providing system level integration using the concept of parallel manufacturing [4, 5]. Parallel Manufacturing (PM) implies the partitioning of interconnect between the semiconductor and the substrate/packaging. The key idea in SHOCC technology is that long and lossy on-chip interconnects can be transferred to thicker and wider off-chip interconnects on the SHOCC substrate. The SHOCC substrate provides high density interconnects for both intra- and inter-chip connections. This is different from the Multi-chip Module (MCM) technology, which uses traditional ICs and provides only inter-chip connections. Only chips designed for the SHOCC substrate can be used in the SHOCC system. Figure 2.1 shows the concept of the SHOCC technology. A bare die is flip-chip bonded onto a SHOCC substrate, which provides metal layers through which signals can be interconnected.

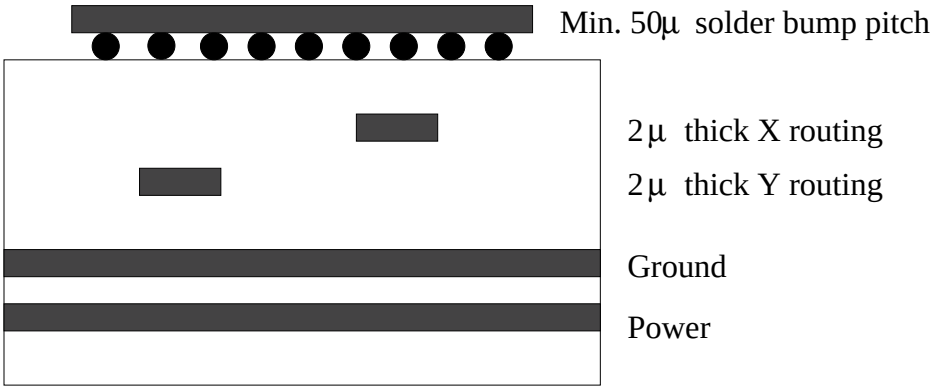


Figure 2.1: Concept of the SHOCC technology

2.1.1 Packaging and Memory Intensive DSP systems

Multi-Chip Module systems have many advantages over conventionally packaged systems. Apart from making the system much more compact, MCM systems are faster due to lower inductances and have lower crosstalk noise [6]. MCMs are especially suited for some high performance signal processing applications which require a considerable amount of computation power with minimal weight and volume [7]. For instance, Gdula et al. [8] developed a multi-chip module system using a high density interconnect technology that reduced the area requirements by over 15 fold. Their system used four Texas Instruments TMS320C25 digital signal processors. The layout rule used 1.5mil conductors at a pitch of 4mils on two layers. Scannel et al. [7] built a multi-chip module DSP system using 16 DSP Processors (TMS320C30s) with each DSP having its own local memory and an interface to a shared memory. They obtained a throughput of 533MFlops with their design. Similar DSP multi-chip module systems have been designed by [9, 10]. [11] used multiple FPGA chips in a multi-chip module system and demonstrated an FIR filter. Dehkordi et al. [12] developed a MCM module consisting of a DSP processor along with some memory and an FPGA chip to provide for other generic logic. Rozier et al. [13] designed an 8,192-point MCM FFT processor using two chips, one of which computes the FFT and the other is a random access data storage element.

Yoshizawa et al. [14] designed a multi-media signal processor using 128-bit wide 16Mb embedded DRAM. Balmer et al. [15] developed a single chip multimedia video processor using embedded DRAM along with off-chip memory. Embedded DRAMs allow fast memory accesses to take place which is important as memory operations are the limiting factor in the performance of DSP systems. However, there is a limit on the amount of memory that can be used in a system using embedded DRAMs. For high data volumes, off-chip DRAM chips have to be used. Most Digital Signal Processors architectures in literature have not used a large number of DRAM chips to increase the memory bandwidth.

2.2 Hardware Architectures of the FFT

The discrete Fourier Transform of a finite duration sequence is given by:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j(2\pi/N)nk} \quad k = 0, 1, \ldots, N-1 \tag{2.1}$$

which can be rewritten as

$$X(k) = \sum_{n=0}^{N-1} x(n) W^{nk} \quad k = 0, 1, \ldots, N-1 \tag{2.2}$$

where, W^{nk} are the twiddle factors. A direct computation of the DFT requires $O(N^2)$ multiplications and additions. FFT algorithms compute the Fourier transform in $O(logN)$ by breaking the original sequence into shorter sequences. Figure 2.2 and Figure 2.3 show the flow graph for radix-2 eight-point FFTs. The vertical nodes in the flow graphs can be thought of as representing storage registers in memory. Both these flow graphs represent *in-place* computation of the FFT which means that data is read and written back to the same set of registers. *In-place* algorithms require N complex registers, while an algorithm that is not in-place would require $2N$ such registers. Figure 2.4 shows a constant geometry

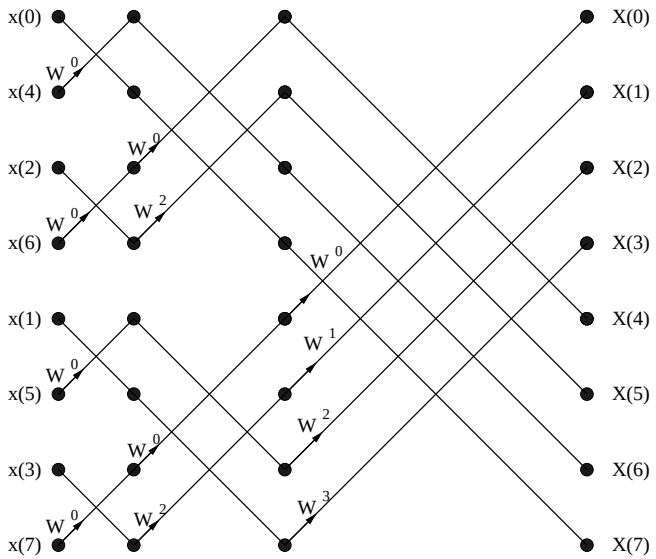


Figure 2.2: Eight Point decimation-in-time FFT

algorithm which is a not an in-place algorithm. Although this approach takes up more memory, it has the advantage of simpler hardware implementation.

The organization and architecture of the FFT processor is usually dictated by performance and cost issues. Different types of organizations of the FFT processors are discussed next [16, 17].

Sequential Processor The basic sequential processor consists of a processing element (PE) that can compute a butterfly. The same memory can be used to store the data, intermediate results and the twiddle factors. The amount of hardware involved is very small and it takes $(N/2)log_2N$ sequential operations to compute the FFT.

Cascade Processor To improve the performance of the sequential processor, parallelism can be introduced by using a separate arithmetic unit for each stage of the FFT. This increases the throughput by a factor of log_2N when the different units are pipelined.

Parallel Iterative Processor By adding more hardware to the sequential processor in each stage, performance can be improved even further. The butterflies can then be computed in parallel in any stage. For instance, adding 3 more PEs in each stage for

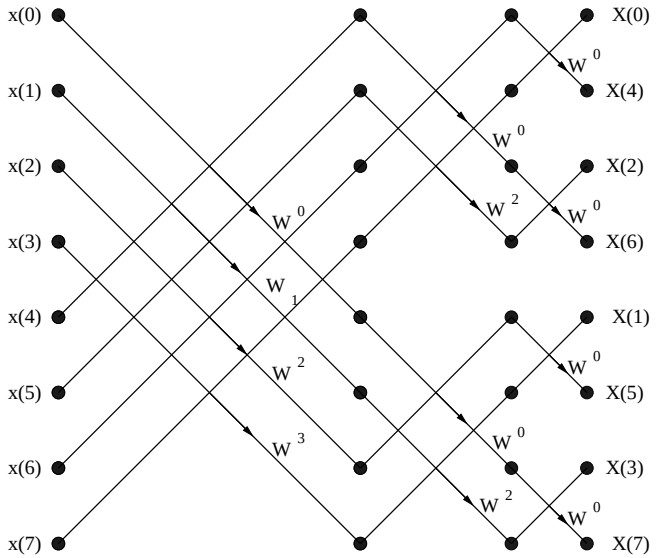


Figure 2.3: Eight Point decimation-in-frequency FFT

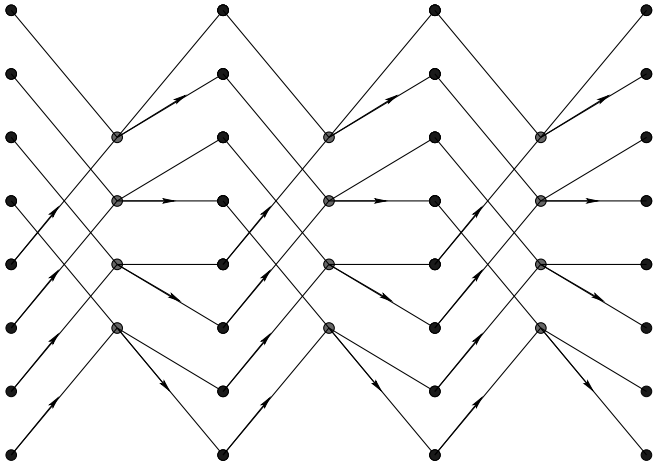


Figure 2.4: 8-Point Constant Geometry FFT

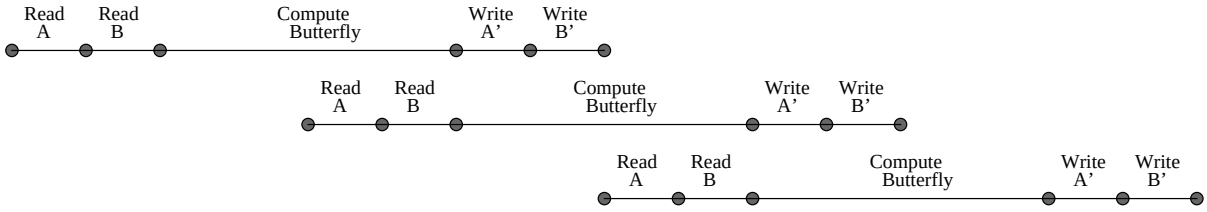


Figure 2.5: Timing Diagram for an "optimum" radix 2 FFT

Figure 2.2 and Figure 2.3 can improve the computation time for each stage by a factor of 4. The total execution time for the parallel iterative processor is $log_2 N$ cycles. Bungard et al. [18, 19] used this scheme in their FFT processor for radar signal processing.

Array Analyzer A fully parallel structure can be constructed by having a PE for each of the butterfly operations. This involves a lot of hardware and is not an attractive option for a large N.

Apart from these basic forms of hardware architectures, different parallel processing schemes can be implemented to improve the performance of the FFT computation. Some of these schemes are: [20]

Time Overlap of Memory and Arithmetic functions The idea behind this is to match the memory and arithmetic operations, such that the system is both memory and arithmetically limited at the same time [21]. The timing diagram for such an optimum system is shown in Figure 2.5.

Use of Fast Scratch Memory Using fast memory for storing all of the data is an expensive option. By using some fast scratch memory some amount of speed gain can be obtained. The idea here is to go a few stages by storing the intermediate results in a fast scratch memory, before having to make an access to the main memory.

Higher Radix Structures A radix 2 butterfly consists of a single complex multiplication

and two complex additions, while a radix 4 butterfly consists of three complex multiplications and eight complex additions. By arranging radix 4 hardware such that the three complex multiplications take place simultaneously, a 4:1 speed increase can be obtained. Bernard et al. [22] used a higher radix FFT structure in their design and used a twiddle factor shifting scheme to maintain the simplicity of lower radix structures.

Pipeline FFTs For a radix r pipeline FFT, $log_r N$ separate hardware butterfly computations proceed in parallel. This is usually used in high-speed designs [23–25]. He et al. [26] have listed different kinds of real-time pipelined FFT structures.

Split Radix FFT The split-radix FFT involves fewer multiplies than the corresponding Cooley-Tukey algorithm. It does require more butterfly computations [27]. For a radix 4 butterfly in the split radix algorithm, two outputs advance through two stages of radix 2 while the other two advance only through one stage.

A lot of work has been published over the last five decades on FFTs. In reference [28], the authors have investigated the design of universal FFT processors to compute multidimensional FFTs. The multidimensional matrix is a matrix-vector product which can always be factored into a chosen form. Only the initial permutation and the twiddle factors change as the dimension changes. The internal data flow, represented by permutation matrices, remains the same. Therefore, a single hardware design can compute one, two or three dimensional FFTs. A distributed architecture comprising of processing elements (with local memory) connected via an interconnection network was used.

A memory efficient implementation of the FFT has been suggested in [29]. By alternating between the Decimation in Time (DIT) and the Decimation in Frequency (DIF) implementations, they require only 2N complex memory locations as compared to 3N in other schemes. The basic idea of the scheme is as follows: if the inputs to a

DIF-FFT are in the natural order, then the output is in the bit-reversed order. As the outputs are being read out in the bit-reversed order, output memory locations start getting free in the bit-reversed order and can be used to store the next set of inputs which are then processed in the DIT mode.

Lee et al. [30], presented new DSP instructions and their hardware architecture for high-speed FFT. These instructions are different from the MAC (Multiply and Accumulate) which existing DSP chips use. The new instructions exploit the data flow in the FFT and include instructions like AMPY (Add and Multiply), MDAC (Multiply and Double Accumulate) and ADMPY (Add and Double Multiply). The authors also propose an architecture which supports these instructions and is faster than existing DSP chips. In [31] an architecture independent SIMD vectorization of the FFT algorithm was done. Function like C macros are provided within the C language where each macro translates to a single SIMD instruction. This is useful for designing different FFT architectures.

From a real-time processing point of view, pipeline architectures are fairly popular. Bi et al. [32] proposed a modified FFT algorithm which makes the hardware simpler. They achieve this by splitting a radix-r butterfly into r simplified radix-r butterfly operations. He et al. [33, 34] designed a 1024-point FFT pipeline processor using radix-2^2 algorithm. The radix-2^2 algorithm has the same multiplicative complexity of radix-4 algorithms but has a signal flow graph similar to radix-2 algorithms. Widhe et al. [25] have used a similar approach to reduce the computational complexity of their radix-8 FFT architecture. By factoring the DFT matrix, all computations can be brought down to the complexity of a radix-2 algorithm. In [35], the authors have designed a low power, radix-4 pipeline FFT processor. To reduce power dissipation, the storage element (SRAM) is split into 4 blocks, which can then be clocked at one quarter of the original frequency. The power supply voltage can then be decreased to reduce the total power dissipation.

Hui et al. have suggested an interesting FFT architecture where the algorithm-to-architecture mapping has been developed from first principles [36, 37]. By

reorganizing the transformation matrix, they are able to exploit regularity of the FFT structure. Their architecture consists of repetitive blocks of radix-4 computation arrays, twiddle multipliers and data commutator circuits. The regularity of the blocks lead to an easier fabrication process.

Yamashita et al. [38] used wafer-scale integration to implement a parallel 16-bit, 8-pt complex FFT. The FFT processor consists of individual repeatable blocks which contain the processing element and the interconnection network. Each processing element consists of a MAC unit and built-in self-test circuits. After block self-diagnosis, active blocks are connected using a programmable contact hole mask.

Various people have used CORDIC techniques to compute the FFT. This makes the hardware simple since no multipliers are required. On the other hand, implementations using CORDIC algorithms take longer as more cycles are required for any computation. Despain [39] showed how to use CORDIC techniques to compute radix-16 FFTs. Angles for different twiddle factors are calculated and implemented in a radix-16 pipeline cascade architecture. Wu et al. [40] have suggested some useful modifications to the CORDIC technique to speed up computation. In their Modified Vector Rotational CORDIC (MVR-CORDIC) scheme, reduction in the number of iterations can be achieved by skipping some of the micro-rotation angles (which can reduce the residue angle error) and repeating some micro-rotation angles.

Fault-Tolerant designs have also been suggested in literature. Li et al. [41, 42] have proposed a C-testable FFT processor design. Only 20 test patterns are required to cover all combinational single cell faults and interconnect stuck-at and break faults. The hardware overhead in their design was 4% for 16 bit numbers regardless of FFT network size. Lu et al. [43, 44] proposed an M-testable design with a 16% overhead for the module level design. Their design uses radix-4 implementation and the N-point butterfly network can be made M-testable by swapping the outputs of the lower left cells of each 4-point module.

2.2.1 Memory Addressing Schemes in FFT systems

A number of approaches have been suggested for the organization of data in memory to make efficient use of available parallel capacity. Pease [45] found that the parities of the input addresses to the radix-2 butterfly are different. The inputs combined in the *r*th stage are those whose ordinals differ in the k_{n-r}th bit. Therefore, for a two memory system, a term can be assigned to the first memory if its parity is 0 and to the second memory if the parity is 1. In this way, memory required for any butterfly is always accessed in parallel from the two memories.

Cohen [46] showed that by reordering the sequence of operations for each stage of the FFT, the *j*th butterfly in the *i*th iteration is $< s,t >$ where

$$s = ROTATE_n(2j,i) \qquad n = log_2 N$$
$$t = ROTATE_n(2j+1,i) \qquad i = 0,1,\ldots(n-1) \quad j = 0,1,2,\ldots(N/2-1)$$

where $n = log_2 N$. This makes the design of the memory controller very simple.

Johnson [47] suggested an efficient hardware implementation for in-place radix-4 FFT algorithm. Bus barrel shifters are used to connect the butterfly inputs and outputs with the memory banks. The address generation circuit is simple in design and uses counters and multiplexers.

In [48] the authors extended Cohen's idea [46] that the inputs in the *p*th pass differ in the *p*th bit. By removing the *p*th bit from the address generation, parity calculation can be avoided. In [49] they suggested an efficient implementation that reduces the complexity of the address generation by about 50%. They extend Cohen's idea [46] and partition the memory further into 4 banks. The butterfly inputs are placed in two memory banks determined by $0b_{n-2}$ and $1b_{n-2}$ and the outputs are stored in two memory banks determined by $b_1 0$ and $b_1 1$, where the butterfly counter is $B = b_{(n-2)} b_{(n-3)} \ldots b_1 b_0$. By removing the *p*th bit from the address at pass *p*, the number of barrel shifters required can

be reduced by half.

Hidalgo et al. [50] have presented efficient architectures for the constant geometry FFT algorithms. The FFT is expressed as a string of operators that are easy to translate into hardware. Each stage of the FFT is equivalent to applying the operator string $B\Gamma$ to the input sequence, where B is the butterfly operator and Γ is the perfect shuffle operator.

Lo et al. [51] have investigated several FFT implementations based on the single butterfly architecture. The memory addressing is formulated as a conflict graph where the vertices represent addresses and the edge indicates that the data corresponding to the addresses should be in different memories. A set of graph coloring rules can then be applied to generate memory addresses.

Harper [52, 53] has suggested stride dependent memory storage schemes that can allow conflict-free access to a vector that uses a constant, predetermined stride. For a vector of size $N = 2^n$ and a stride $S = 2^s$, he suggests memory storage schemes for two cases. When ($s < n$), row r is rotated $(r) mod 2^s$ places relative to its state in the interleaved scheme. When ($s \geq n$), blocks of contiguous rows are rotated relative to the preceding blocks.

2.2.2 Twiddle Factor Generating Schemes in FFT systems

Most FFT processors use lookup tables to store twiddle factors. Some FFT algorithms use schemes (eg. prime factor algorithm which requires N to be split into mutually prime factors) that eliminate the need for twiddle factor multiplications [54].

Cohen [46] suggested a simple way to generate coefficient addresses. He showed that the jth butterfly in the ith iteration requires the kth power of $W_0 = e^{(-2\pi i/N)}$, where k can be obtained from j by masking out the $(n-1-i)$ least significant bits. This fact can be used to design simple control structure using counters.

Hasan et al. [55] have suggested an interesting method to generate coefficient addresses. Their scheme uses a k-bit counter which is partitioned into two sections. The less significant section of $b = log_2(N/2)$ bits keeps track of the coefficient index and the

more significant section keeps track of the stage of the FFT. The lower b-bits are used to generate a coefficient address array and the remaining bits are used as the control to a MUX which selects the appropriate index.

CHAPTER 3

FFT Architecture

3.1 Introduction

This chapter discusses in detail the architectural and algorithmic approach used in the design of the FFT processor. The main focus in this chapter is the organization of the FFT system and the memory management and twiddle factor generation schemes used in the design. The following chapter focuses on the signal integrity issues related with the design.

3.2 Physical Architecture

The physical architecture of the FFT system is shown in Figure 3.1. The chip set contains 1 GB of memory distributed among 128 64Mbit DDR-2 DRAM chips, and four custom 1 sq.cm. micro-accelerator chips. The FFT is designed as a radix-64 engine, with two micro-accelerator chips working together in each stage. Each of the chips therefore reads 32 complex numbers during each FFT computation.

For a 0.25μ technology, it is possible to design a 32-bit multiply/accumulate unit in

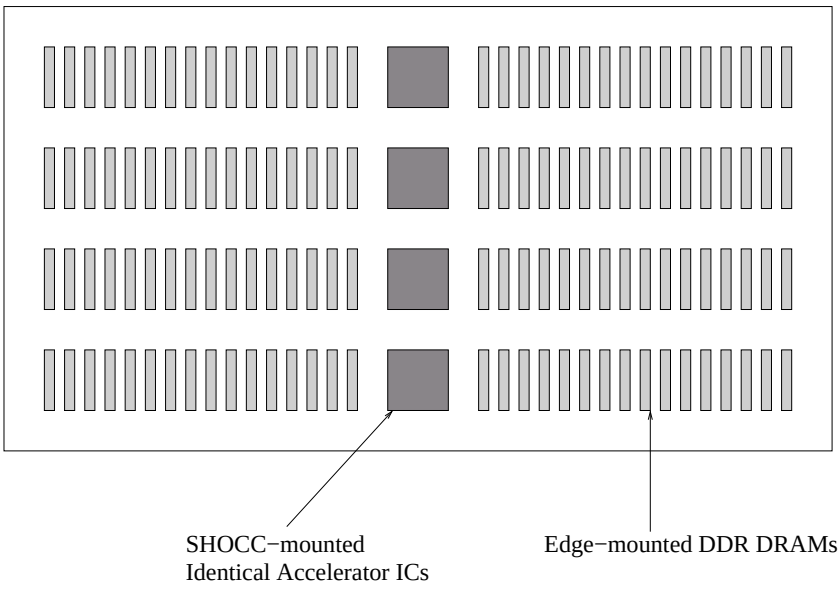


Figure 3.1: Physical Architecture of the FFT processor

less than $1mm^2$. For a 0.18μ technology, the area requirement would be even less. Therefore, it is possible to accommodate a large number of floating-point units on each chip. Each micro-accelerator chip, in our design, contains an array of 64 32-bit multiply-accumulate units and 32 memory interface channels. Each of the 60-pin DRAMs are edge-mounted onto a high density 8cm x 8cm SHOCC interposer substrate using a previously developed solder-bump edge mounting technique [56]. This edge-mounting technique permits chips to be mounted at low cost as no special processing steps are involved in the assembly. The only limitations are pin count (50 per cm of edge) and power dissipation (around 0.3 - 0.4 W due to the limited cooling path). Each memory chip is wired directly to one (and only one) memory port on a micro-accelerometer chip i.e. there is no shared memory bus, and each DRAM has its own bus. The high density substrate thus contains 128 independent 16-bit memory buses, which together with the control and inter-accelerator bus make up approximately 8000 total nets to be accommodated. This large amount of wiring is possible in the SHOCC technology, as it contains two signal, a power and ground layer in the substrate, with the signal layer being routed down to 20μ pitch. Each micro-accelerator has 2500 signal pins, requiring a solder

bump pitch of 140μ, again made possible with the SHOCC technology.

Each micro-accelerator chip performs floating-point operations to contribute to the 64-point FFT. A million-point FFT performed in Radix-64 requires 4 stages to complete. To avoid bus contention for memory reads and writes we split the memory into two sets. In the first stage of the FFT, the data is read from the first set of memory and written to the second set. In the next stage, data is read from the second set and written to the first set. This "ping-pong" action cuts the memory conflicts that would arise if data were read and written to the same set of memory. The 64-point FFT is further broken down in Radix-8 and hence requires two stages to compute. The first two chips perform the first stage of the FFT and the results of this stage are passed to the next two chips, through the high speed bus. A 64-point FFT in Radix 8 would require 32 8-point units and these are split evenly between all the chips. Therefore, each of the two micro-accelerator chips in the first stage read in 32 numbers each and perform 4 8-point FFTs. After twiddling the results, the output is passed onto the next two micro-accelerator chips where the sequence of operations is similar. The only difference is that the twiddle factors in the final twiddling stage are different for the chips in the first stage and those in the second. The breakdown of these operations is shown in Figure 3.2. By pipelining the operations as shown, a 64-point FFT can be computed every 20ns. This is assuming that all the 64 numbers required for the FFT can be read in 20ns.

The computation block uses data read from the DRAMs and the on-chip SRAM to compute the FFT. Details of these operations are described later in this chapter. Apart from the floating point multipliers and adders, the arithmetic block also contains units for swapping and negating data. These are used in the first two stages of the 8-point FFT, which involve twiddling with $\pm$j.

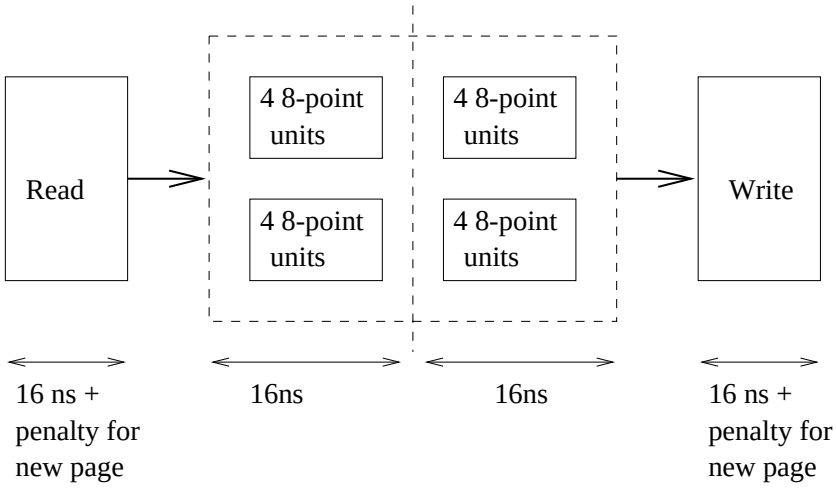


Figure 3.2: Schedule of Operations for the entire 64-point FFT

3.3 Optimal FFT Algorithm

This section addresses the two main bottlenecks in the design of the high performance million-point FFT system. In the first part, we outline a scheme for the storage and retrieval of data in DRAMs that minimizes the timing overheads of DRAMs and makes full use of all the memory bandwidth available in the design. In the second part, we address an issue specific to an FFT, that of handling the twiddle factors. By making full use of all available arithmetic units, the memory bandwidth requirement can actually be reduced.

3.3.1 Memory Addressing Scheme

The key to a successful high volume, high performance FFT system lies in efficient memory management. In our scheme, presented next, we stagger the results obtained from the previous stage before storing them in memory. The amount of stagger is not constant between the FFTs. This places the data in the correct memories for the next stage. The algorithm is not in-place, i.e. data is not stored in the same location from which it was read, but in a different order in another set of memory. This allows us to efficiently

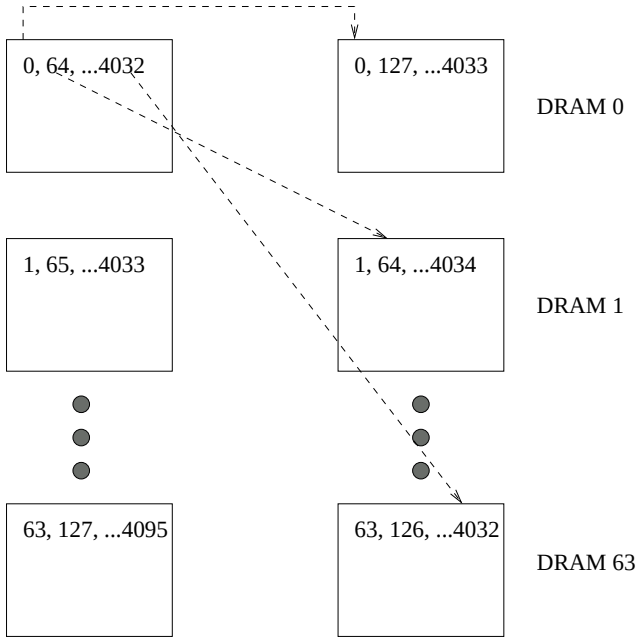


Figure 3.3: Memory Layout of Data after the First Stage

pipeline the READs and WRITEs to memory. Our scheme is closest in nature to the scheme in [52, 53] where data in memory is mapped according to the stride of the algorithm. The main difference is that since the stride is not constant between different stages of the FFT, an additional rotation is provided before data is stored in order to maintain the same stride. In addition, our memory mapping scheme also takes care of the DRAM related issues like the precharge/refresh times.

For the Radix-64 case, the memory mapping scheme can be extended and is given by the following relation:

$$DRAM\_No = (FFT\_No + index)\% \ 64 \tag{3.1}$$

where, $FFT\_No = \lfloor (index/64) \rfloor$ and *index* refers to the index number of the data ($0 <$ index $<$ 1,048,575)

The memory allocation for any stage is given in Figure 3.3. This scheme introduces a stagger of 64 required in each stage and can be easily implemented using shift-registers at the input and output.

3.3.1.1 Memory Mapping of Data into different DRAMs

The above scheme would work perfectly if an ideal memory were used i.e. if a particular data from any row/column address could be read in the given time. This is not the case with conventional DRAMs where the row precharge times can add significant timing overheads. For our design we have used DDR SDRAM (MT46V4M16) from Micron Semiconductor Products Inc. [57]. The main DRAM candidates for high speed and high bandwidth application are the DDR DRAM, Rambus DRAM and SLDRAM [58]. SLDRAM is based on the SyncLink technology [59]. The SLDRAM memory system consists of Memory Controller, internal bus that is split into CommandLink and DataLink and SLDRAM memory devices. The projected peak data rate for the SLDRAM is 800Mb/s. However, SLDRAMs are still in the development phase and off-the-shelf components are not available as yet and for this reason not used in our design. The DDR DRAM was chosen over the Rambus DRAM for four reasons [60]. The first is that the design required a large number of memory channels to run at reasonable speeds. Rambus DRAMs are good when running small data buses at high speeds. Second, the memory controller was to be integrated on the processor chips and on-chip estate was of crucial concern. The memory controller for the DDR DRAM is comparatively much less complicated than the Rambus Controller. Third, the Rambus die size is larger (10-20% [61]) than the DDR DRAM die size due to additional interfaces needed, and had an impact on the off-chip estate as well. Finally, for long Rambus buses, it is difficult to maintain the impedance and other parameters within their strict specifications to be able to run the bus at high speeds.

The DDR SDRAM can read or write data at both the rising and falling edge of the clock, which makes it faster than other SDRAMs. In the chosen DDR SDRAMs, a random access request takes 60ns to serve (compared with less than 5ns for modern SRAMs). Thus if the data accesses were all truly random the total memory bandwidth of our system would be 15Gb/s and the FFT performance unacceptably low. A block

Table 3.1: Some of the timing parameters associated with DDR SDRAM

Parameter	Value
ACTIVE to ACTIVE/ AUTO REFRESH Command Period	60ns
ACTIVE to READ or WRITE Delay	15ns
ACTIVE bank A to ACTIVE bank B command	15ns
ACTIVE to PRECHARGE command	45ns
PRECHARGE Command Period	15ns

diagram of the DRAM is shown in Figure 3.4. The total memory is divided into 4 banks and each row consists of 4096 bits. Once a row in a particular bank has been activated, data from that row can be read or written every 5ns. However, a minimum burst size of 2 is required for this. Another bank in the same memory can be activated while the first bank is still active after a time of 15ns. The various relevant timing parameters have been summarized in Table 3.1.

To minimize the overheads due to precharge times, bank, row and column addresses have to be generated in a fashion that hides the row activation time. The state diagram in Figure 3.5 shows the relative timing of the active and read/write commands. Each state takes 5ns and a single 64-point FFT can be completed in 20ns. The reads and writes are always in bursts of 4 and once the read/write command is issued data appears on the data bus every 5ns (after an initial latency of 2 clock cycles). The key point to note here is that an active command can be given to another bank while data is being read/written to the previous bank. In our scheme, a 64-bit complex data is written in 4 adjacent columns and hence data is always accessed in a burst of 4. An extra NOP state is required between the active command and the read/write command. This is because commands can be registered only at the positive clock edge, even though data could appear on the data bus on both edges of the clock.

The scheme for generating bank and row addresses is different for reads and writes and is summarized below:

Reads:

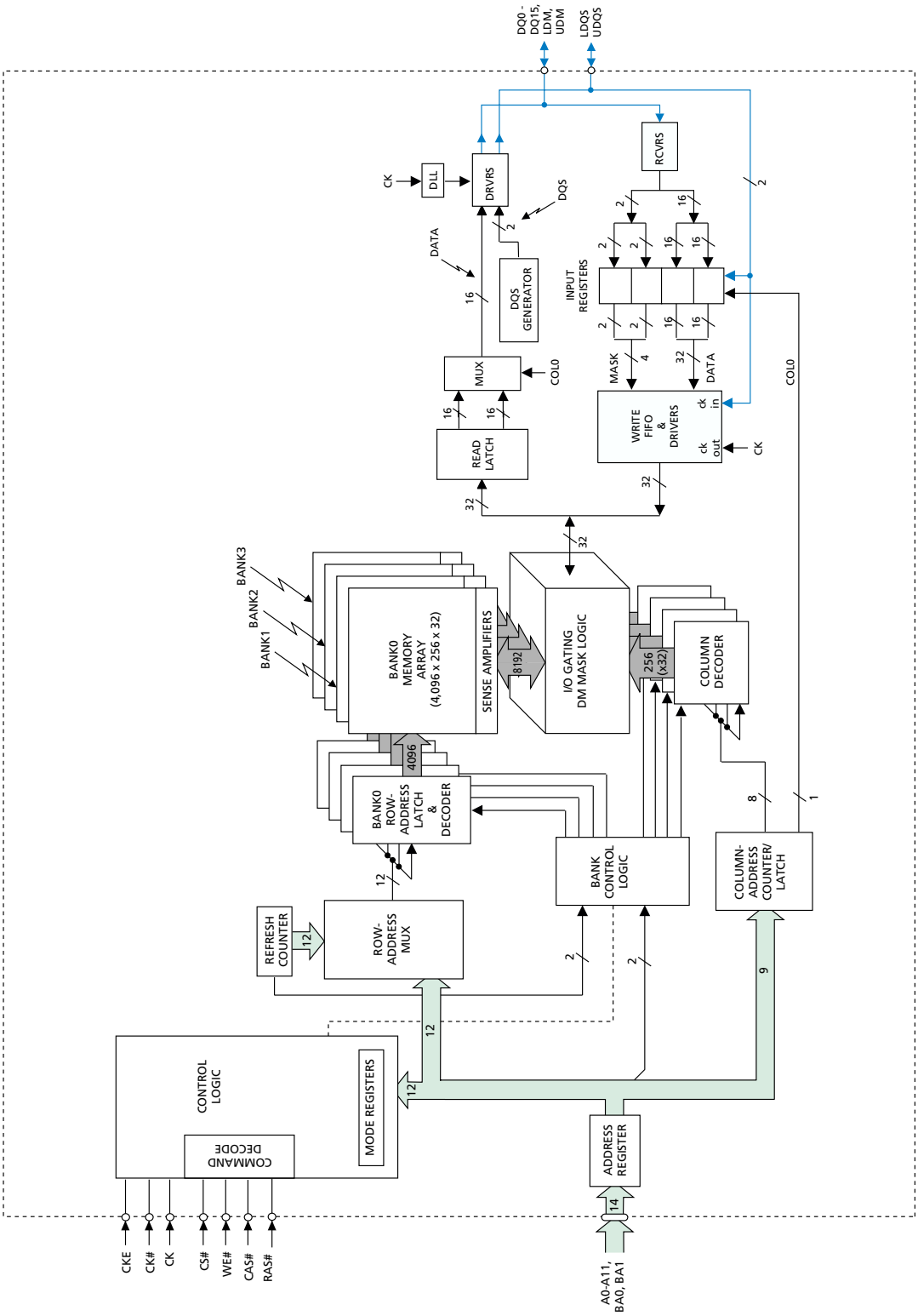


Figure 3.4: Functional Block Diagram of DDR SDRAM (MT46V4M16) (http://www.micron.com)

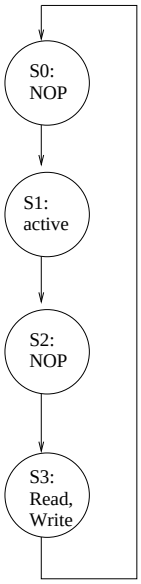


Figure 3.5: State Diagram Showing the Timing of Different Commands

$$Row\_No = \lfloor FFT\_No/4 \rfloor \tag{3.2}$$

$$Bank\_No = FFT\_No \,\%\, 4 \tag{3.3}$$

Writes:

$$Row\_No = \lfloor FFT\_No/256 \rfloor \tag{3.4}$$

$$Bank\_No = (\lfloor FFT\_No/64 \rfloor) \,\%\, 4 \tag{3.5}$$

where $FFT\_No = \lfloor (index/64) \rfloor$ as before. A row hop takes place after every 4 FFTs while reading and after every 256 FFTs while writing. A bank hop takes place after every FFT while reading and after every 64 FFTs while writing. Figure 3.6 shows the arrangement of data in the 4 banks after writing at the end of the first stage. In stage two, the underlined data is read from each memory for the first FFT. Different column numbers need to be generated for the different DRAMs but the data is always read in bursts of 4

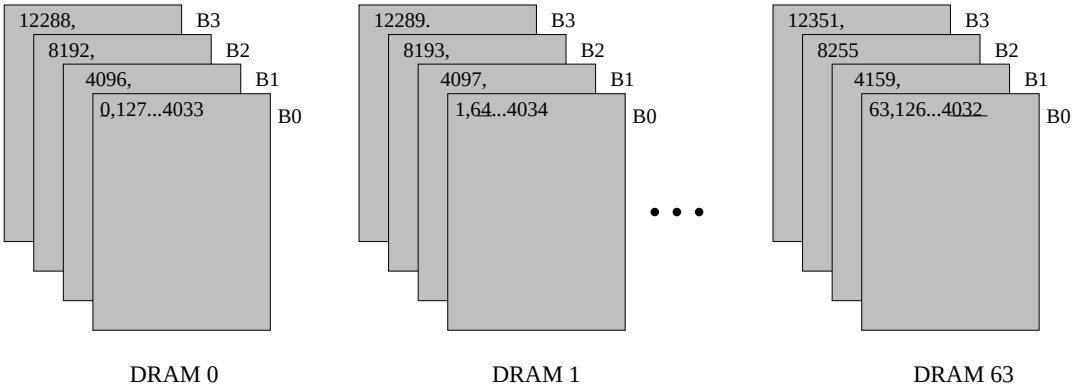


Figure 3.6: Allocation of Data in Different Banks in Memory after the First Stage

and this maintains the ideal throughput.

To determine the feasibility of this scheme, we implemented a synthesized design in Verilog. The timing diagram from the simulations is shown in the following figures for both reads and writes. This corresponds to data being written after the end of the first stage and data being read for the second stage. Simulations show that it is indeed possible to read and write data at the maximum possible rate of DRAM without incurring any of the overheads associated with DRAM operation. Figure 3.7 shows the timing waveforms of different signals during the read operation. The waveforms correspond to the clock, RAS, CAS, WE, Addr, Ba, DQ and the current state (for the FSM in Figure 3.5) respectively, from top to bottom. In state 1, an active command is given to the next bank to be accessed (seen from Ba for state 1). In state 3, the read with auto precharge command is given for the current bank and data appears on the data bus after a latency of 2 clock cycles. Before the read operation, the four banks of the memory had data 0000, 1000, 2000 and 3000 respectively. Data is first read from bank0 (indicated by 4 bursts of 0000) and then the bank is switched and data 1000 is read from bank1 and so on. After 4 FFT cycles, an active command is given to the next row in bank0 (indicated from 001 on the Addr waveform) and the cycle can be repeated. Figure 3.8 shows the same waveforms for the write phase. In this case, the data is written to the same bank for 64 FFTs and then a bank change takes place. The memory mapping scheme outlined above ensures

continuous flow of data to and from the DRAMs. This makes the performance of the FFT system as good as a system with SRAMs used in place of the DRAMs.

3.3.2 Twiddle Factor Generation Scheme

The other point of consideration in the design of the FFT system is the handling of the twiddle factors. For most small DSP systems, the twiddle factors can be stored in on-chip memory [13, 62]. This works fine for smaller systems but would require huge amounts of memory for large systems like the one under consideration. For the million-point FFT under consideration, a million twiddle factors would have to be stored in memory. This would double not only the physical memory requirement but also the number of memory channels. The other alternative, which we discuss next, is to generate the twiddle factors needed for the FFT on the chip itself. As discussed before, computing an FFT in radix 64 is equivalent to breaking down one row of numbers into a two-dimensional array of 64 rows. This can be expressed as [63]

$$X(s,r) = \sum_{m=0}^{M-1} W^{Lmr} W^{ms} \sum_{l=0}^{L-1} x(l,m) W^{Msl} \tag{3.6}$$

where W^{ms} is the twiddle factor required to twiddle the 64-point FFT results for the next stage computations and is given by

$$W^{ms} = e^{-j(2\pi/N)ms} \tag{3.7}$$

For a Radix-64 FFT, L is 64 in (3.7) for the two dimensional representation. The sequence of operations is then

1. Compute the 64-point FFT of each column
2. Twiddle the results with W^{ms}
3. Compute the FFT of each row

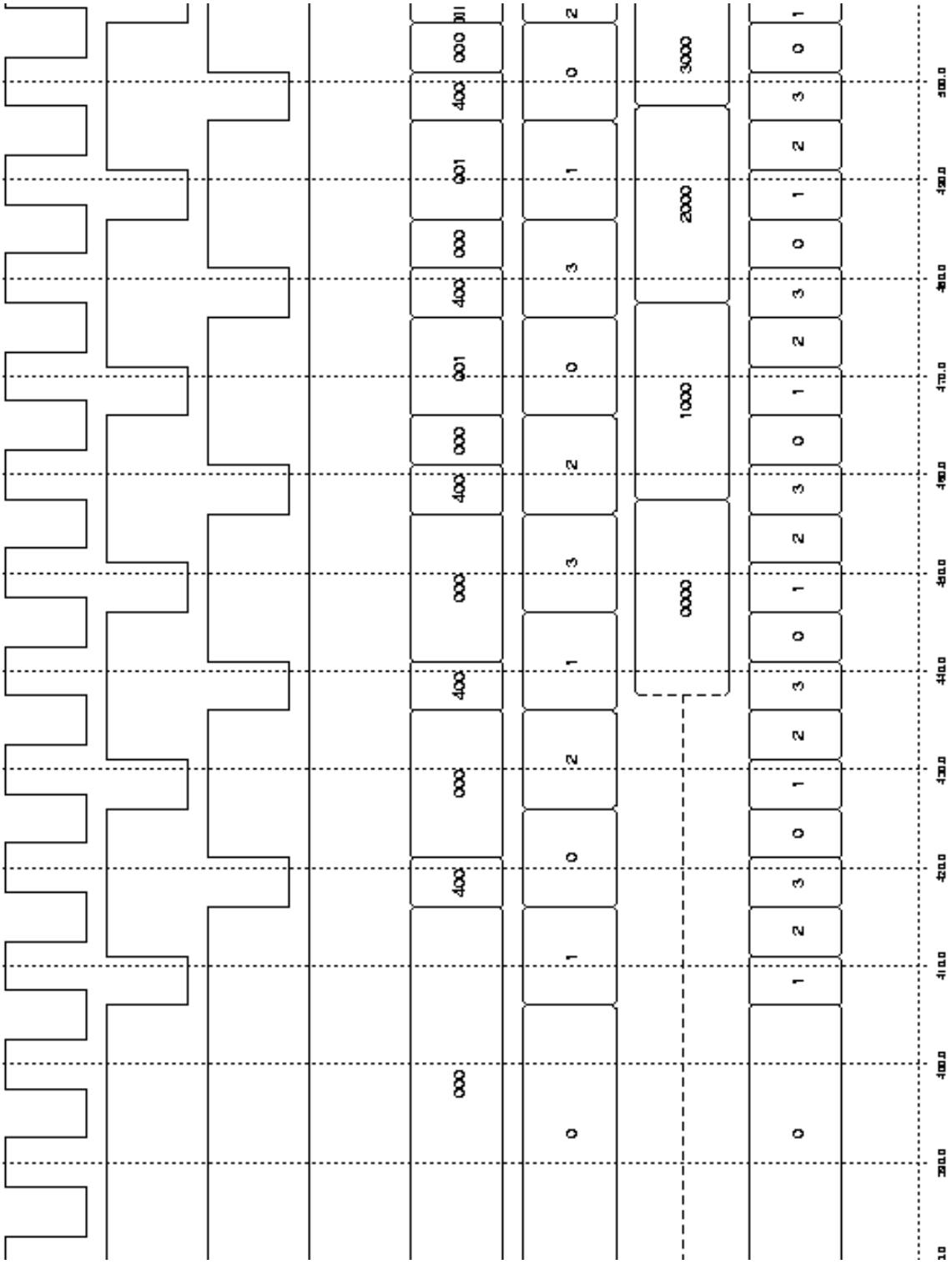


Figure 3.7: Clk, RAS, CAS, WE, Addr, BA, DQ, CurrentState Waveforms for the Read Phase

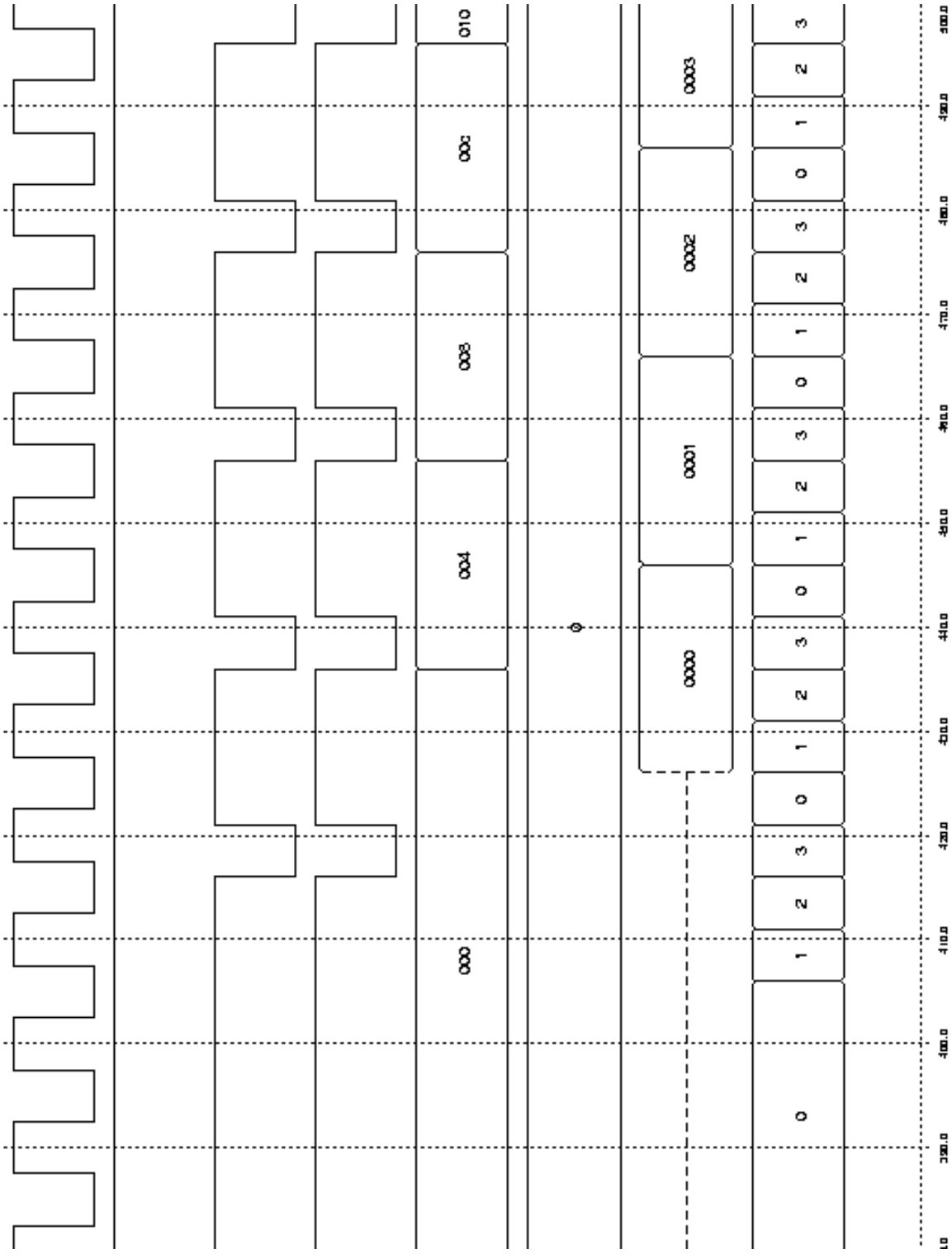


Figure 3.8: Clk, RAS, CAS, WE, Addr, BA, DQ, CurrentState Waveforms for the Write Phase

For the twiddle factors, s varies from 0 to 63 in all stages while W^{ms} takes on values depending on the stage. For our twiddle factor generation scheme, we store an initial 64 twiddle factors (corresponding to m=1 in the previous equation) in on-chip SRAM. Twiddle factors for the any FFT can be generated from the twiddle factors of the previous FFT and the initial twiddle factor set, since

$$W_N^{ms} = W_N^{(m-1)s} W_N^{1.s} \tag{3.8}$$

The initial twiddle factor set for the next stage can be generated in the previous stage by storing the twiddle factor values corresponding to m=64, as

$$W_N^{64.s} = W_{N/64}^{1.s} \tag{3.9}$$

In this way, with a relatively small amount of on-chip memory, all the twiddle factors can be generated on-chip and no extra memory bandwidth is required. This assumes that the twiddle factors for the next FFT can be computed in the previous cycle, along with the computations of the FFT. For FFT systems, this is possible because the first two stages of the 8-point FFT computation do not require any multiplications and the 64 32-bit floating point multipliers can be used in generating the twiddle factors for the next stage. The sequence of operations is shown in Figure 3.9. All the computations required for the FFT can be completed well within the budget of 20ns.

3.4 Performance Comparison with Conventional Off-The-Shelf Systems

The FFT engine described above computes a 64-point FFT in 20ns. A million-point FFT can be computed in 4 stages with 16,384 64-point FFTs in each stage. The total time to compute the million-point FFT is therefore 1.31ms. The performance capability of the

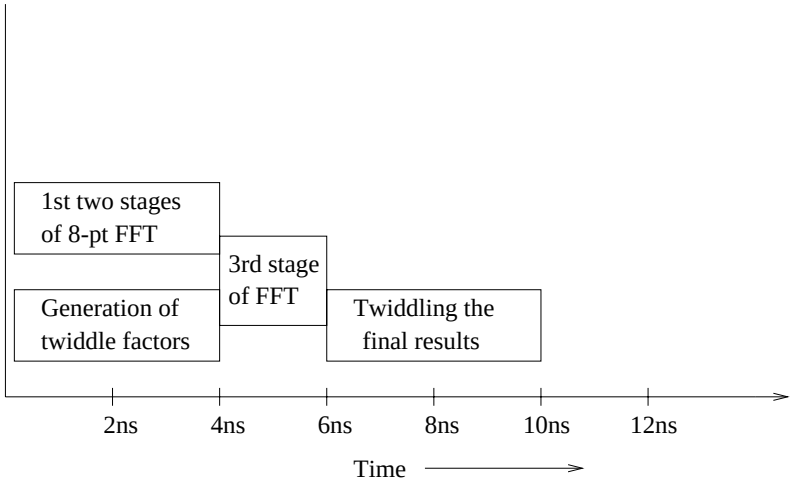


Figure 3.9: Sequence of operations for the FFT engine

system is as follows:

1. Peak performance of $2.56x10^{11}$ FLOPs (if all arithmetic units were 100% occupied)
2. Peak Memory bandwidth of 47.6GBps (i.e. if all DRAMs were operating continuously in burst mode)
3. Sustained floating point performance of $8.64x10^{10}$ corresponding to 763 million-point FFTs/second
4. Sustained memory performance of 47.6GBps

We compared the performance of our FFT engine with two other systems. The first one uses four BOPS Inc. DSP chips [64]. The system has 4 32-bit memory channels. Each chip has 4 PEs and each PE has 5 FP units. This system would take approximately 80 sq.cm of PCB and would perform a million-point FFT in 21.5 ms, which is more than an order of magnitude slower than ours. We also compared our engine with a G4 Velocity Engine implementation of the FFT, using Motorola's Altivec technology. Computing a million-point FFT on their system takes 511ms, almost 400x slower than ours [65].

CHAPTER 4

Signal Integrity Analysis using SHOCC substrates

This chapter looks at various circuit related aspects such as the substrate cross-section, number of routing layers, routing pitches etc. that maximize the bandwidth between two die interconnected via a SHOCC substrate. The I/O bandwidth is defined as follows:

$$I/O\ Bandwidth = (No.\ of\ Wires) * (Bit\ Rate)$$

The number of wires and the bit rate depend on the physical dimensions of the system and the signal integrity issues. Given a maximum processor die area, the bump diameter and the bump pitch limit the number of solder bumps that can be accommodated. The number of signal layers in the SHOCC substrate determine whether breakout is possible for all the solder bumps. This limits the number of I/O pins that can be put on a chip. Signal integrity issues, like crosstalk noise and attenuation which increase as the length of the signal trace increases, place further routing restrictions. The ratio of the number of signal pins to ground and power pins determines the simultaneous switching noise (SSN). The physical dimensions and noise budget therefore limits the number of I/O pins that can be used. Timing issues like skew and jitter determine the minimum cycle time of the system. The noise budget and the timing budget together determine the maximum data

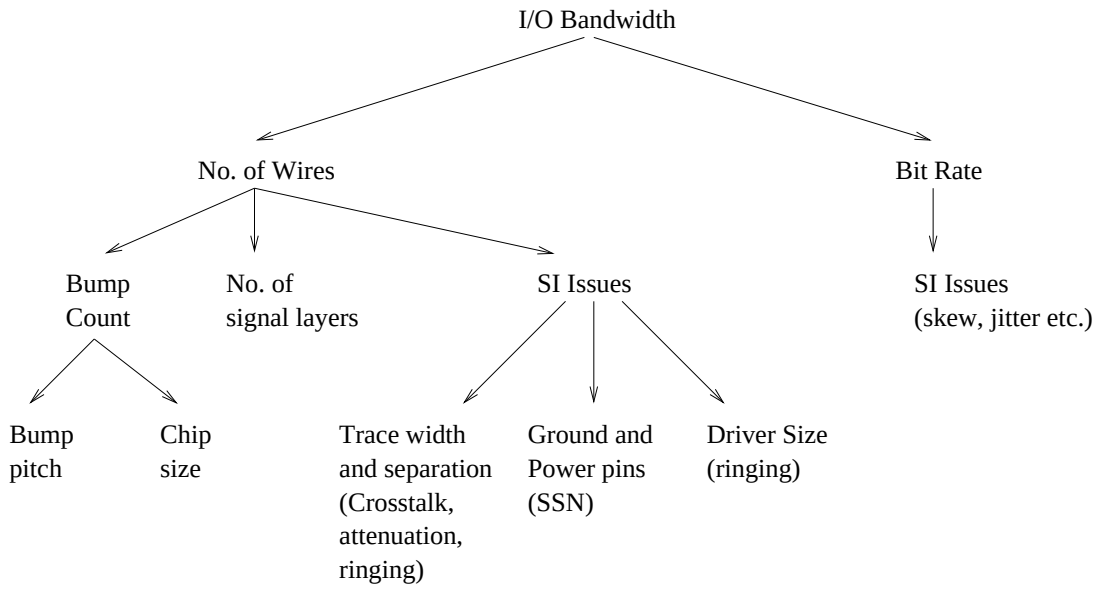


Figure 4.1: Dependence of various parameters on the I/O bandwidth

rate that can be obtained. The dependence of the I/O bandwidth on various parameters is summarized in Figure 4.1.

The overall bandwidth can be obtained by performing a noise and timing analysis to determine the number of wires and the bit rate respectively. The total system noise is composed of the crosstalk noise between signal traces, the SSN, the attenuation and the reflection noise. The total noise can then be approximated as the root of the sum of squares of the individual noise components.

$$V_{Total} = \sqrt{(V_{crosstalk}^2 + V_{SSN}^2 + V_{attenuation}^2 + V_{reflection}^2)}$$

The total noise should fit within the noise margin. For the TSMC 0.25μ technology, the noise margin was found to be 1.04V. In the current design, the total noise of the system has been constrained to be within the 70% of the noise margin, i.e.

$$V_{Total} \leq (0.7) * Noise\ Margin \approx 0.7V$$

The minimum cycle time is governed by the skew and jitter components. An accurate

timing analysis requires constructing eye diagrams taking into account the skew, inter symbol interference and process variations. In this work, we have considered only the skew component which arises due to difference in the signal lengths. The minimum cycle time is given by [66, 67]

$$t_{cy} \geq 2t_u + t_a + t_r$$

where, t_u is the timing uncertainty due to skew, jitter etc., t_a is the aperture time and t_r is the transition time required for each waveform to switch states. The total skew includes the on-chip component as well.

In the current design, only alternatives that use two layers for routing have been considered. This reduces the cost of the substrate processing. Other constraints used in the design, in order to reduce processing costs, are: (a) a minimum trace width of 10μ and (b) a maximum substrate stackup thickness of 50μ. The trace thickness is 2μ in all cases considered.

The number of signal I/Os that need to be routed out, including control signals, in the design is around 2000. To add more flexibility to the design, we have used an estimate of around 2500 signal I/Os to be routed out per chip. In addition, the ratio of power and ground bumps to I/O bumps is assumed to be 1:1. This means that a total of 5000 bumps need to be placed on each chip. For a 1cmx1cm chip, the bump pitch required to support these many bumps is 140μ. A two stage breakout approach has been assumed as shown in Figure 4.2. In the initial phase of the breakout, the routing pitch between the traces in the two layers is small. This phase also contributes the most to the crosstalk noise. In the intermediate phase, the routing becomes XY in nature and mutual coupling (and hence crosstalk) between traces on the two layers becomes very small. In the final phase, the routing pitch is determined by the routing under the DRAMs and is larger than the pitches in the first two phases. This phase contributes least to the total crosstalk noise. To determine the routing pitches as shown in Figure 4.2 we have also assumed an overhead of

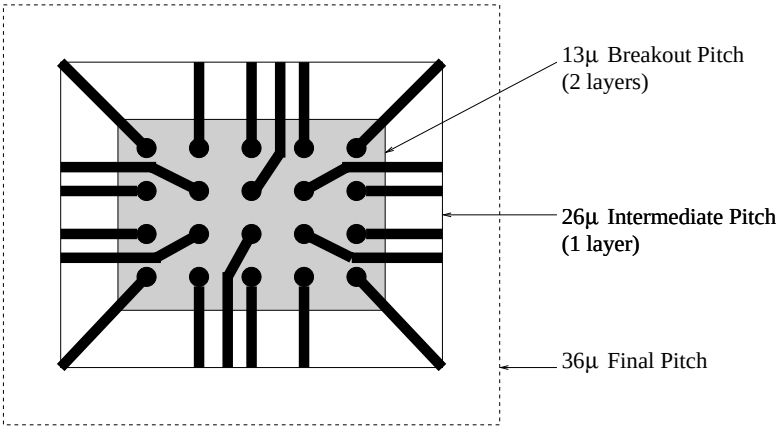


Figure 4.2: Two Stage Breakout Routing Approach

15-20% to account for the loss of area due to power and ground vias.

4.1 Modeling and Simulation

SHOCC structures were modeled using Ansoft's Maxwell Q-3D Parameter Extractor [68], which uses finite element analysis, method of moments and multipole expansion techniques to extract R, L and C parameters. Once the R, L and C parameters were determined, a transmission line model of the SHOCC lines was simulated in HSPICE to determine the crosstalk noise and delays.

4.1.1 Description of the Circuit

A typical signal path from a driver on one chip to the receiver on another chip is shown in Figure 4.3. The signal starts from the output of driver and goes through ≈0.2mm on-chip line. It then travels through a via to the solder bump on the long off-chip SHOCC line. At the other end of the SHOCC line, it goes through another solder bump, a second 0.2mm of an on-chip segment and finally to the receiver. The on-chip segments account for the distance that the signal has to travel from the driver/receiver to the nearest solder bump. This value of the on-chip segment length was taken from [69, 70]. The on-chip R, L and C values were obtained from [4] and the R, L, C parameters for the solder bump were taken

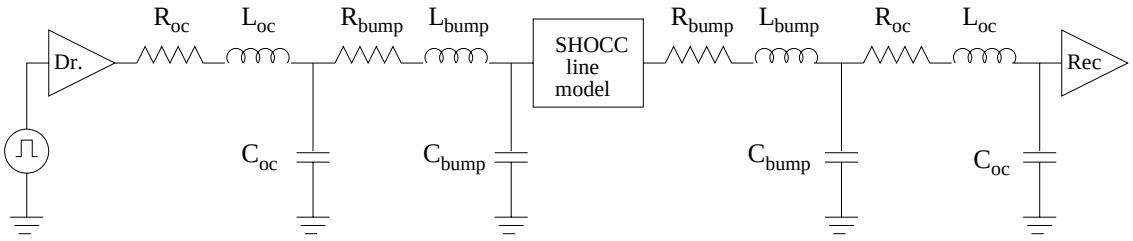


Figure 4.3: Equivalent Circuit Representation of the SHOCC system

Table 4.1: R, L, C values for on-chip segment

Resistance $R_{on}\ (\Omega/cm)$	Capacitance $C_{on}\ (pF/cm)$	Inductance $L_{on}\ (nH/cm)$
166	1.75	3.19

from [70]. These values are shown in Table 4.1 and Table 4.2.

Long off-chip interconnects can be modeled in three possible ways [71].

1. If the signal rise time (t_r) is more than twice the round-trip propagation time ($2t_0 l < t_r$), then the interconnect can be modeled as a lumped circuit.

2. A lossy transmission line where the line resistance is much larger than the characteristic impedance ($R_0 >> Z_0$), can be modeled as a distributed RC line.

3. A slightly lossy line ($R_0 << Z_0$) can be represented as a cascade of RLC segments.

Most long SHOCC interconnects fall into the last category. To analyze the crosstalk noise, the model used in [69] was extended to include the couplings from all the nearest

Table 4.2: R, L, C values for the solder bump

Resistance $R_{bump}\ (\Omega)$	Capacitance $C_{bump}\ (F)$	Inductance $L_{bump}\ (H)$
$1.72x10^{-2}$	$1.87x10^{-14}$	$8.49x10^{-12}$

neighbors. Figure 4.4 shows a segment of the SHOCC transmission line model and Figure 4.5 shows a segment which includes the coupling effects of the nearest neighbors. The various parameters in the figure are:

- R = Resistance per unit length of a trace
- C = Self Capacitance per unit length of a trace
- C_{mtt} = Mutual Capacitance per unit length between two neighbors on the top layer
- C_{mtb} = Mutual Capacitance per unit length between a trace on the top layer with its neighbor on the bottom layer
- C_{mbb} = Mutual Capacitance per unit length between two neighbors on the bottom layer
- L = Self Inductance per unit length of a trace loop
- L_{mtt} = Mutual Inductance per unit length between two neighbors on the top layer
- L_{mtb} = Mutual Inductance per unit length between a trace loop on the top layer with its neighboring loop on the bottom layer
- L_{mbb} = Mutual Inductance per unit length between two neighboring loops on the bottom layer

The minimum number of segments required to model a transmission line are governed by the following two equations [71]

$$T_r \geq 3.5\pi T_0 l \tag{4.1}$$

$$R_0 l \leq 0.1 Z_0 \tag{4.2}$$

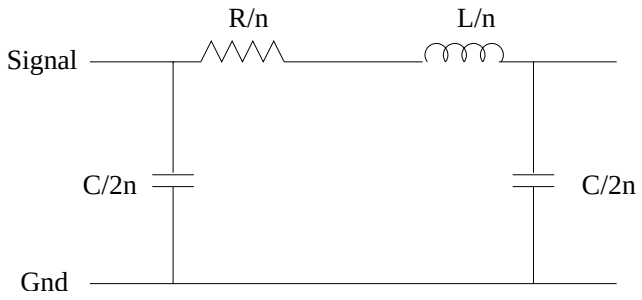


Figure 4.4: Equivalent Circuit Representation of a segment of the SHOCC line

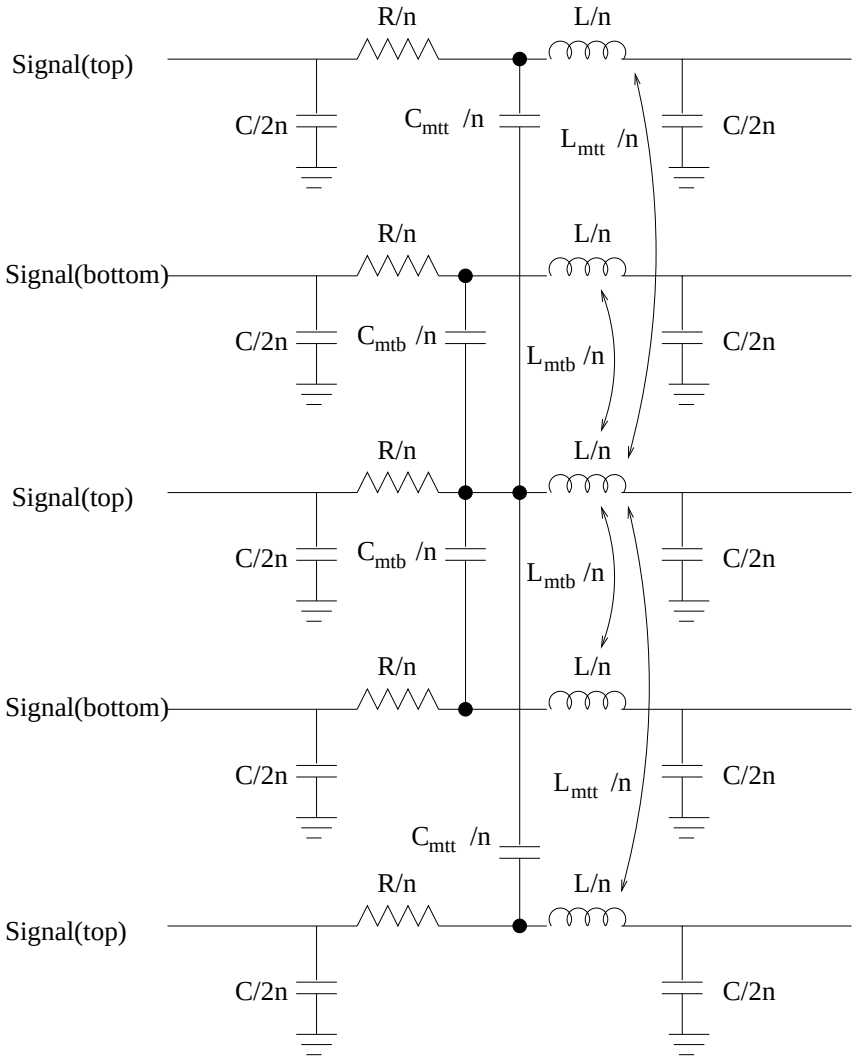


Figure 4.5: Equivalent Circuit Model used for Crosstalk Simulations

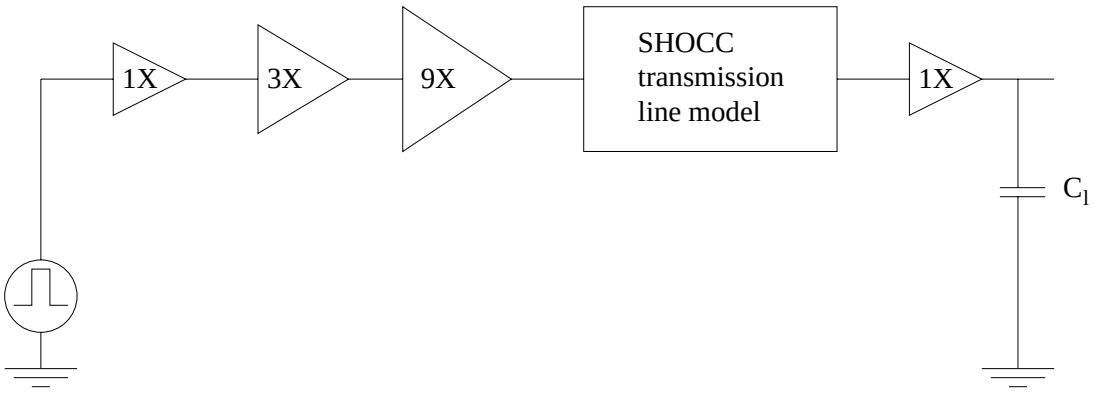


Figure 4.6: Cascaded CMOS Drivers of Increasing Size

Our simulations use a rise time of 80ps and for Z_0 in the range of 50-100Ω, 15-16 segments are sufficient to model long transmission lines. In our simulations we have taken 20 segments to model all transmission lines, where each segment is represented using the π model. The on-chip interconnects and the solder bump were modeled as lumped circuits, using an L model [72] and the complete SHOCC line was then driven by cascaded CMOS drivers. In order to optimize delay, multi-stage drivers with a constant stage ratio were used. By scaling up each stage by a constant factor u, the delay of each stage is identical and is given by $t_{p,stage} = ut_{p0}$ [73]. The total delay is given by

$$t_p = Nut_{p0}$$

and

$$N = \frac{ln(x)}{ln(u)}$$

The optimal scaling factor can be obtained by equating the derivative of t_p with respect to u to 0 [73]. The optimum u comes out to equal to $e = 2.7182$. We have taken a stage ratio of 3 in our design. Different driver sizes were used to drive the SHOCC line as shown in Figure 4.6. The convention used is the same as in [69] where 1X represents the smallest inverter, 3X a two stage inverter etc.

The channel width (W) and channel length (L) parameters for a two-stage driver for

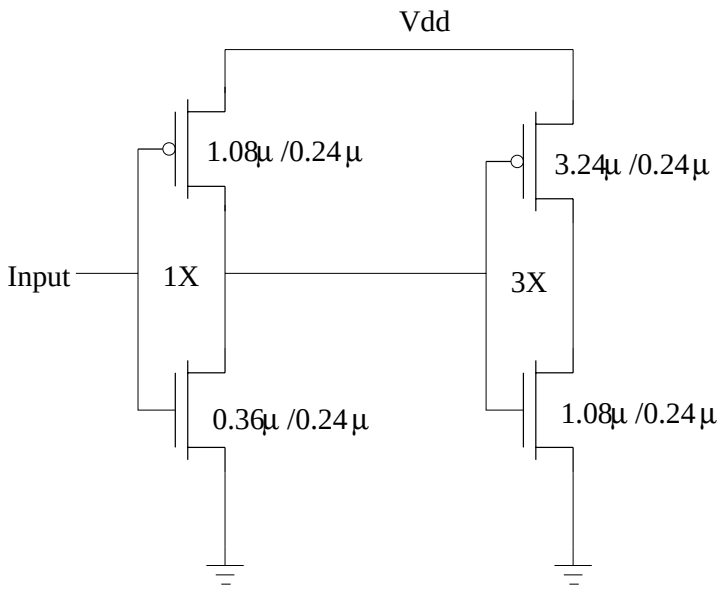


Figure 4.7: A Two-stage Cascaded CMOS driver with a Stage ratio of 3

0.25μ CMOS technology is shown in Figure 4.7.

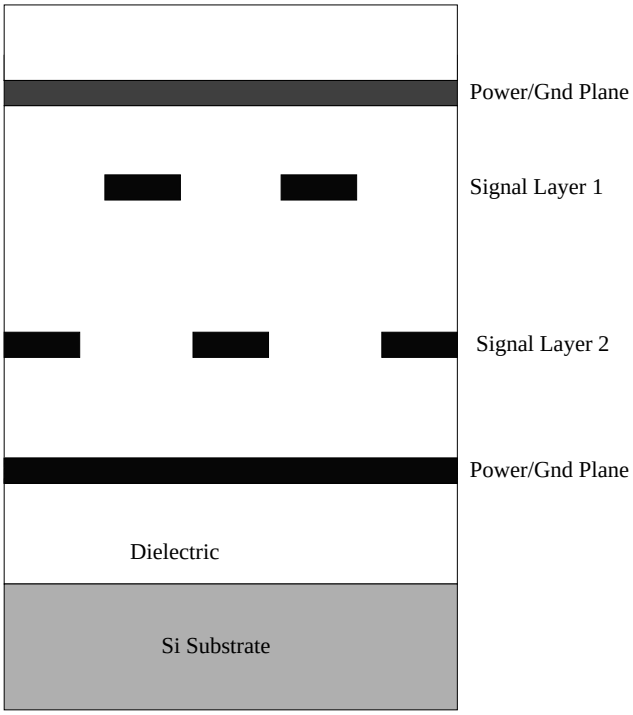
4.2 Substrate Alternatives

In order to accommodate all the signal I/Os with the pitches dictated by physical limits while maintaining the noise and timing constraints, various substrate stackups were explored. The first choice is between an IMPS (Interconnected Mesh Power System) like substrate and more conventional signal/power stickups. In the IMPS configuration, the power and ground lines alternate between the signal lines. For instance, for a two-layer IMPS, all the traces in one layer run in the X direction and all the traces in the other layer run in the Y direction. In each layer, the power and ground traces alternate with the signal traces. The mutual couplings between traces in the two layers is quite small due to the traces being perpendicular to each other. However, the problem with this topology is that the routing densities become half that of conventional approaches due to the presence of the power and ground traces. For our design, this would mean that the pitch in the initial phase would be 6.5μ in two layers or 13μ in each layer. This is not possible with a

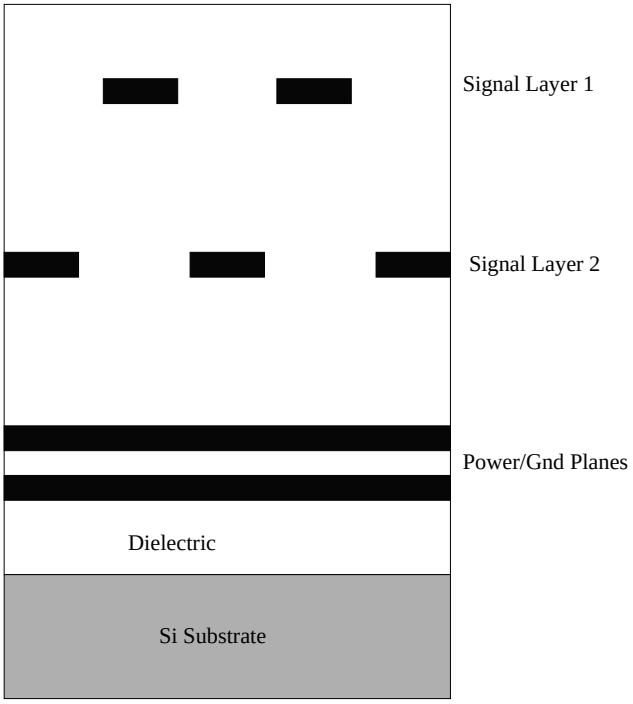
minimum trace width of 10μ, as the minimum pitch that the IMPS can handle is 20μ (with a trace separation of 10μ). Therefore we did not use the IMPS topology in our design. From a routing density point of view, conventional signal and power/ground layers are more attractive and these have been discussed next.

Figure 4.8 shows two possible stackups with two signal layers each that can be used to route the signal I/Os. In the first case, the signal layers are sandwiched between the power and ground planes. The advantage of this stackup is that the delay and noise on the both the signal layers is close to identical. However, signal vias have to cut through the first plane, making routing more difficult. In the second case, the signal layers are on top of the power and ground planes, both of which are placed very close to each other. This maximizes the amount of decoupling capacitance between the power and ground planes. However, the signal characteristics in the two layers are not identical since the layers are at different distances from the power and ground planes. In our design, we have chosen the second substrate stackup to exploit the inherent decoupling capacitance.

For the above stackup, we performed a noise and timing analysis using the approach outlined in the next section. The crosstalk noise in the initial breakout phase of routing comes out to be 0.5V, leaving very little margin for other noise components. This was clearly not acceptable. We therefore modified the substrate stackup to include an additional layer between the two signal layers which acts as a local ground (LG) in the initial routing phase as shown in Figure 4.9 [74]. The addition of the local ground shields the two signal layers from each other and reduces the coupling between the two layers. The crosstalk noise then comes down to acceptable levels as shown in Figure 4.10 where the two curves correspond to traces on the top layer with a driver size of 81x. As can be seen from the figure, considerable reduction in noise can be obtained by adding the local ground. More than two and a half times reduction in noise was obtained for a trace length of 1cm.



(a)



(b)

Figure 4.8: SHOCC substrate stackups

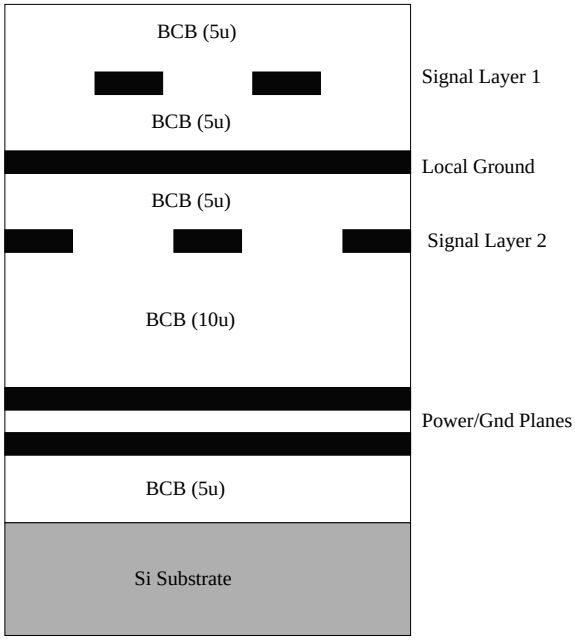


Figure 4.9: SHOCC Substrate with a local ground

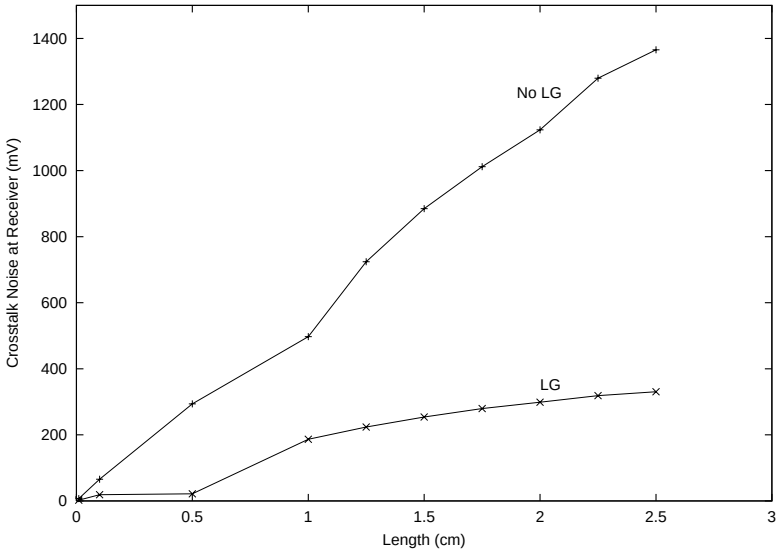


Figure 4.10: Crosstalk Noise with and without local ground (driver size = 81x) for a trace on the top layer (width = 10μ and pitch = 26μ

Table 4.3: Impedance and Propagation Delay for SHOCC lines

Substrate Dimensions (μ)	Signal Layer	Resistance (Ω/cm)	Capacitance (pF/cm)	Inductance (nH/cm)	Characteristic Impedance $Z_0 = (L/C)^{1/2}$	Propagation Delay $\tau = (LC)^{1/2}$
w=10,p=26 (No LG)	Bottom Top	8.62 ”	0.63 0.121	3.65 4.64	76.12 195.8	47.95 23.7
w=10,p=26 (LG)	Bottom Top	8.62 ”	0.94 0.546	3.06 3.75	57.06 82.87	53.63 45.25
w=15,p=36 (No LG)	Bottom Top	5.75 ”	0.687 0.135	4.2 4.85	78.2 189.5	53.72 25.59
w=10,p=36 (No LG)	Bottom Top	8.62 ”	0.59 0.157	3.7 4.6	79.2 171.17	46.72 26.87

4.3 Simulation Results

4.3.1 Impedance and Propagation time

The extracted values of R, L and C and the calculated values of the characteristic impedance and propagation delay for various dimensional parameters are shown in Table 4.3. For all cases, the characteristic impedance of the bottom layer is smaller than that of the top layer while the propagation delay is larger. A few points can be noted from the table. The proximity of the bottom layer to the power/ground planes makes the capacitance parameter larger. It also reduces the loop area, making the inductance smaller. The effect on the capacitance increase dominates over the effect of the decrease in inductance in this case, leading to the above results. The presence of the local ground makes the behavior of the top layer approach that of the bottom layer. The capacitance for both the bottom and the top layer increases, though the increase is more dramatic for the top layer. The difference in the inductances also decreases. A wider trace shows larger capacitance and inductance values (due to a larger area).

Table 4.4: Mutual L and C parameters for SHOCC lines

Substrate Dimensions (μ)	C_{mtt} (pF/cm)	C_{mbb} (pF/cm)	C_{mtb} (pF/cm)	L_{mtt} (nH/cm)	L_{mbb} (nH/cm)	L_{mtb} (nH/cm)
w=10,p=26 (No LG)	0.116	0.042	0.186	1.298	0.63	1.33
w=10,p=26 (LG)	0.0093	0.00317	-	0.565	0.186	0.176
w=15,p=36 (No LG)	0.106	0.012	0.216	0.975	0.315	1.4
w=10,p=36 (No LG)	0.068	0.0068	0.16	0.87	0.34	1.1

4.3.2 Crosstalk Noise

The extracted mutual coupling parameters are listed in Table 4.4. Without a local ground, the mutual capacitance between two traces on the top layer (C_{mtt}) is always greater than the mutual capacitance between two traces on the bottom layer (C_{mbb}). This is expected due to the proximity of the bottom layer to the ground/power planes and therefore a larger capacitance to ground (see Table 4.3). The local ground, as before, tends to make things more even for the bottom and top traces. The mutual capacitance between the top and bottom traces is larger than the mutual capacitance between traces on the same layer. This is again expected, since edge side capacitance is smaller than broadside or diagonally placed traces. The mutual capacitance between the top and bottom layer trace vanishes when the local ground is introduced. Also, the last two rows in the table show that for the same width, a wider trace has higher mutual capacitance than a thinner trace. Mutual inductance between two traces on the top layer is larger than the mutual inductance between two traces on the bottom layer due the larger area of the top layer loops. As before, the presence of the local ground tends to reduce the mutual couplings between traces.

The crosstalk noise for the different cases is shown in Figure 4.11-Figure 4.16. The

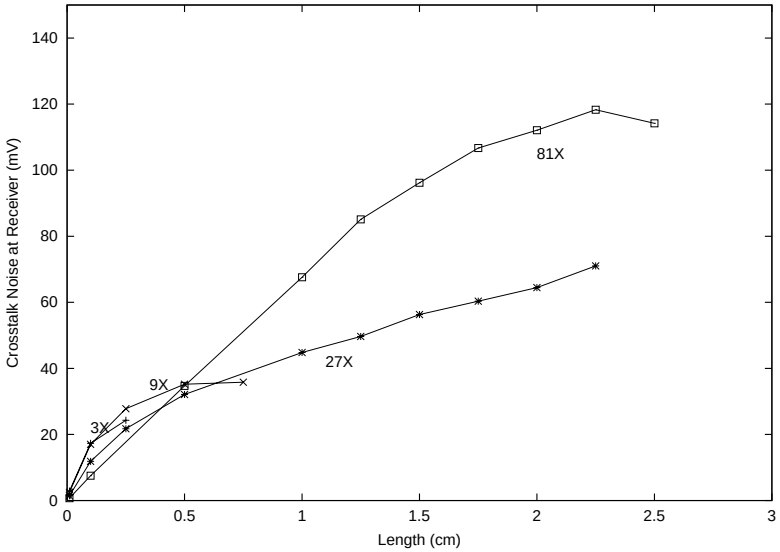


Figure 4.11: Crosstalk Noise for a bottom trace (w=10,p=26,LG)

effect of the stronger couplings for the top layer can be seen in these figures, where the top traces show much larger crosstalk noise than the bottom traces. The local ground is very effective in reducing the crosstalk noise in the initial breakout phase. Noise is reduced by 5-6 times with the addition of the local ground.

4.3.3 Delay

The worst-case delays for the different cases is shown in Figure 4.17-Figure 4.22. The delays for a bottom trace are larger than the delays for a trace on the top layer. This verifies the difference in the propagation times listed in Table 4.3.

4.3.4 Reflection Noise

The reflection noise for a trace in the final routing phase is shown in Figure 4.23. The reflection noise is much smaller than the corresponding crosstalk noise components and constitutes a fairly small percentage of the total noise. The total reflection noise, when all sections are considered together, is 0.006V for the bottom trace and 0.028V for the top trace.

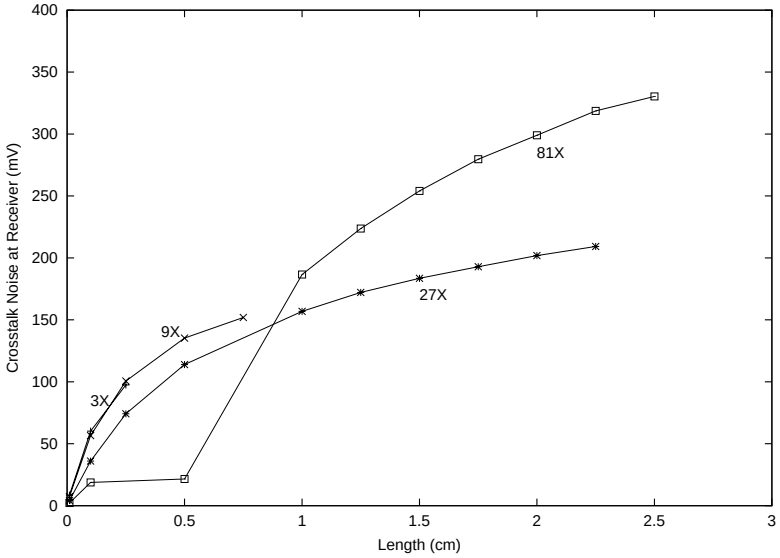


Figure 4.12: Crosstalk Noise for a top trace (w=10,p=26,LG)

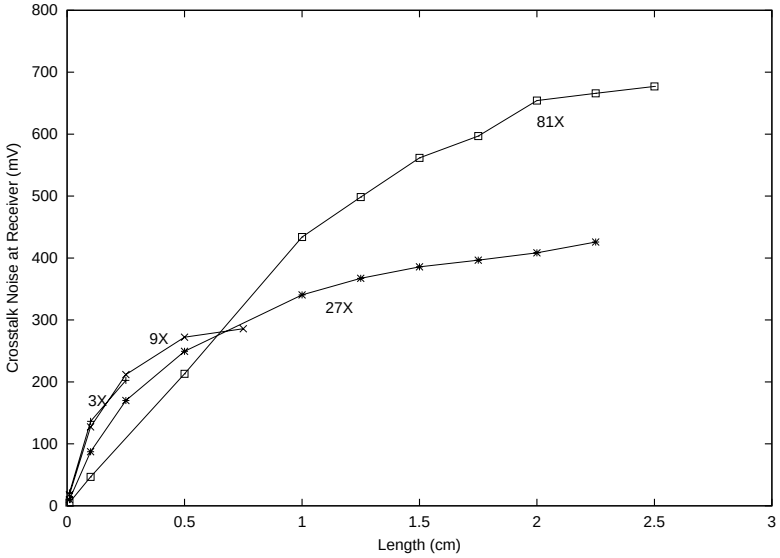


Figure 4.13: Crosstalk Noise for a bottom trace (w=10,p=26,No LG)

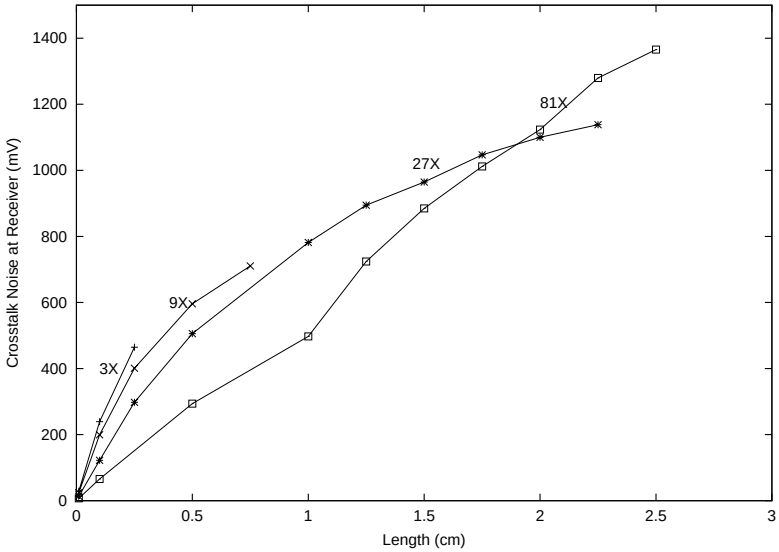


Figure 4.14: Crosstalk Noise for a top trace (w=10,p=26,No LG)

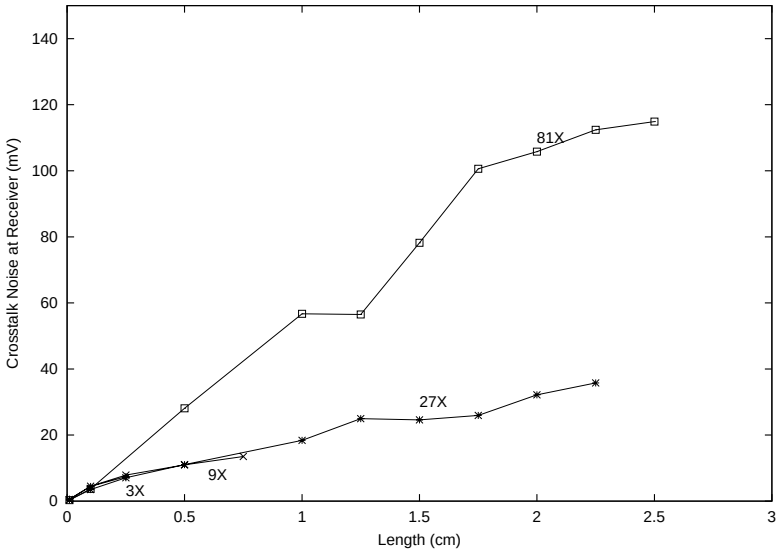


Figure 4.15: Crosstalk Noise for a bottom trace in XY routing mode (w=10,p=36,No LG)

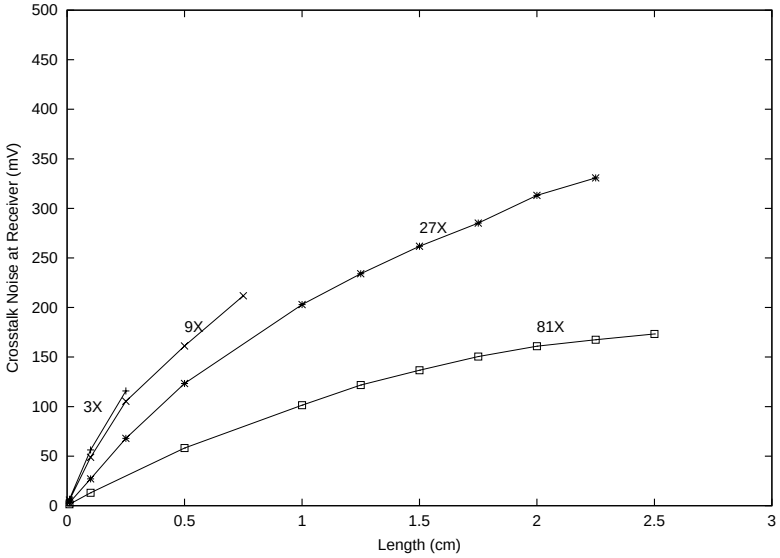


Figure 4.16: Crosstalk Noise for a top trace in XY routing mode (w=10,p=36,No LG)

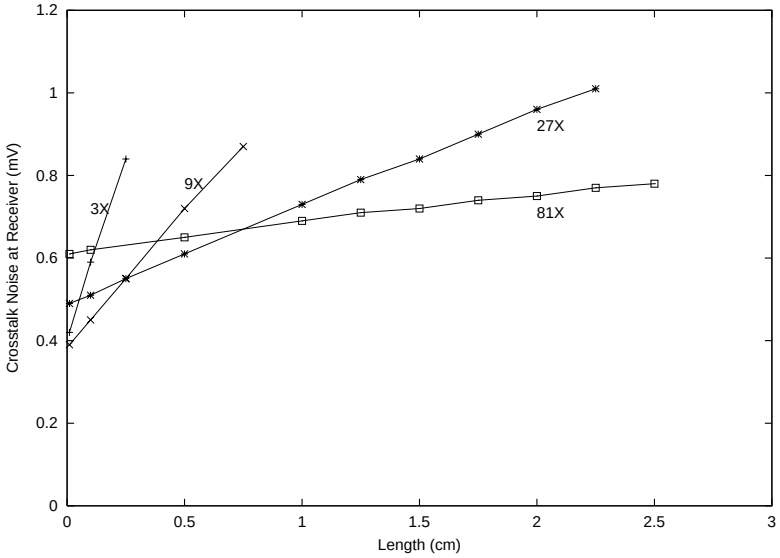


Figure 4.17: Delay for a bottom trace (w=10,p=26,LG)

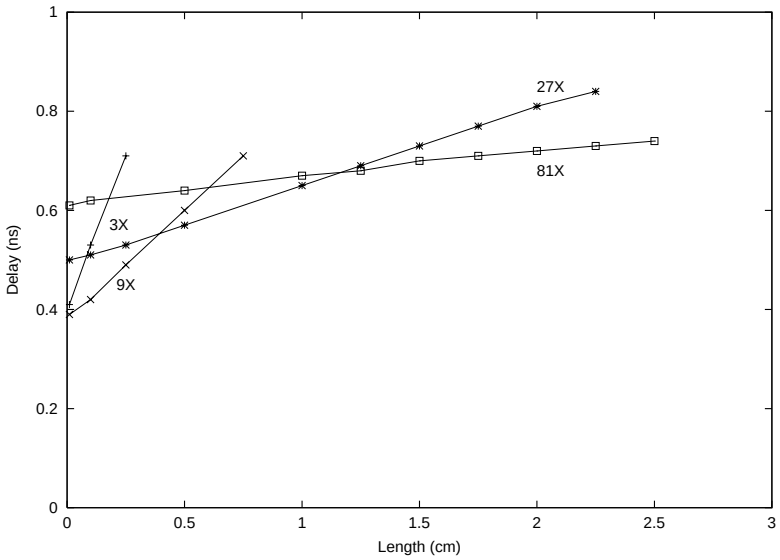


Figure 4.18: Delay for a top trace (w=10,p=26,LG)

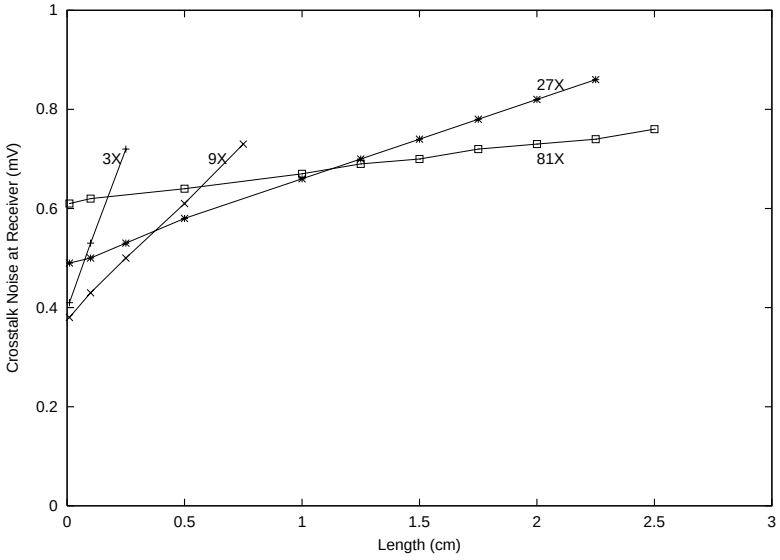


Figure 4.19: Delay for a bottom trace (w=10,p=26,No LG)

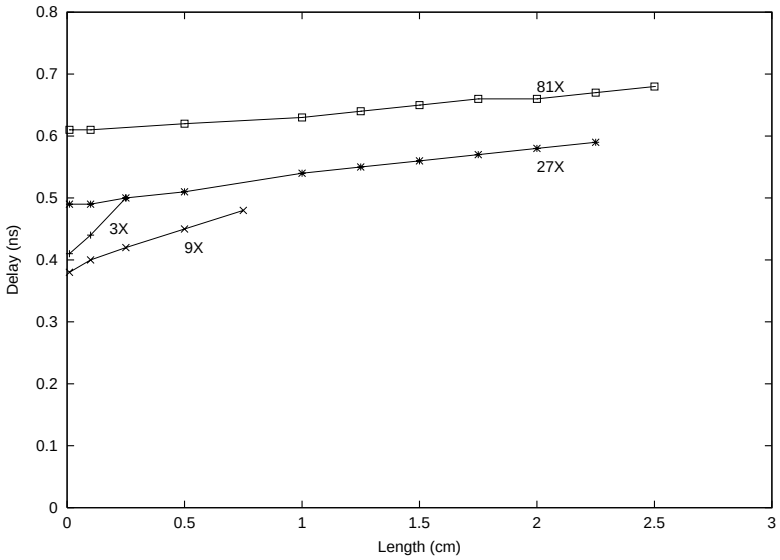


Figure 4.20: Delay for a top trace (w=10,p=26,No LG)

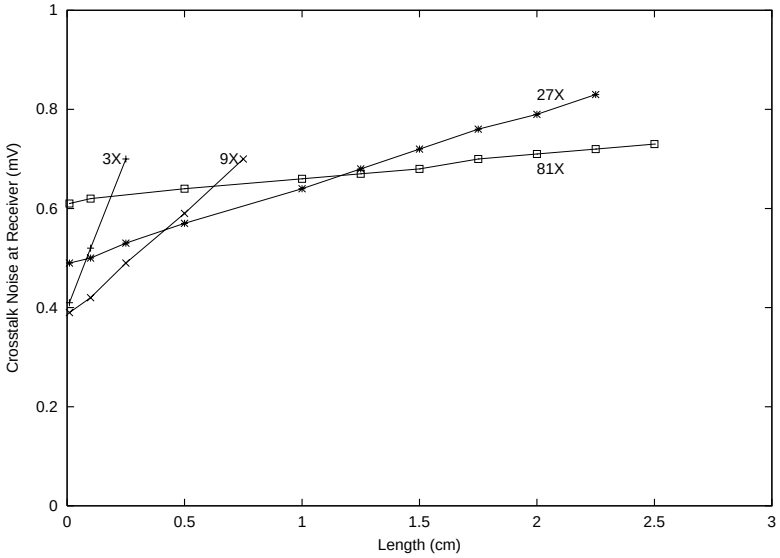


Figure 4.21: Delay for a bottom trace in XY routing mode (w=10,p=36,No LG)

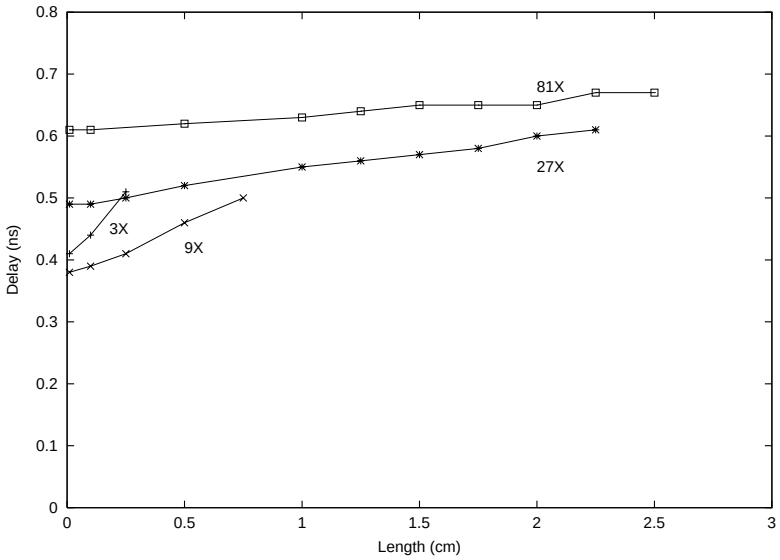


Figure 4.22: Delay for a top trace in XY routing mode (w=10,p=36,No LG)

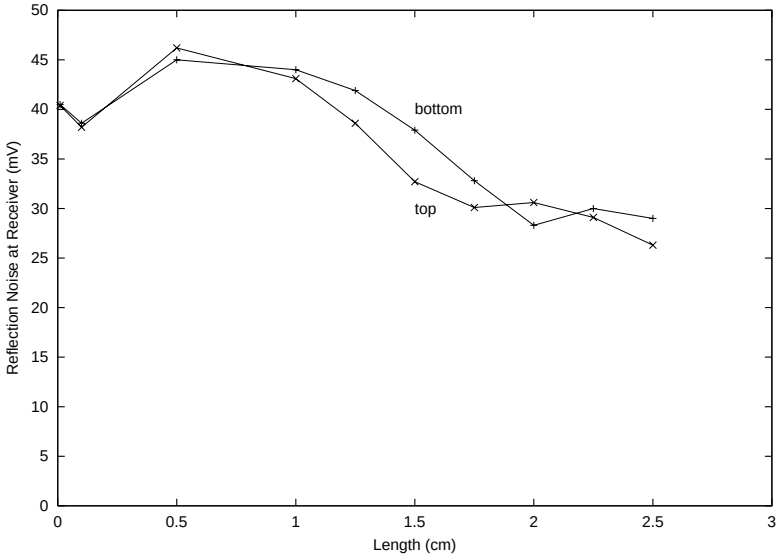


Figure 4.23: Reflection Noise for the final phase routing (w=10,p=36)

Table 4.5: Driver Output Resistances

Driver Size	Resistance (Ω)
3X	1860
9X	800
27X	320
81X	125

4.3.5 Effect of Driver Size

Different driver sizes were used to drive the SHOCC line. Not all drivers were able to drive long SHOCC lines. The maximum interconnect length that could be driven by a driver was determined by slowly increasing the length of the interconnect till the waveform at the receiver became sufficiently distorted. A single stage driver was unable to drive interconnects as small as 0.01cm and has therefore not been included.

Table 4.5 shows the estimated driver resistances. The driver resistance was obtained by connecting the output of the driver to a capacitance of known value and simulating it in HSPICE. The driver resistance can then be obtained from the RC time constant. However, the decrease in driver resistance is also accompanied by an increase in the driver capacitance which has the effect of increasing the overall delay.

Smaller driver sizes tend to make SHOCC lines behave like lossy on-chip lines. This can be seen from the delay characteristics which are similar to those in [70].

4.4 Overall Bandwidth Analysis

The escape length for the first breakout stage is 1cm. The maximum length that a signal has to traverse under the DRAMs, is 2.2cm (for 11 DRAMs in a row at a pitch of 2mm). The second escape length for a total routing length of 4cm is 0.8cm. The total crosstalk noise can therefore be determined by summing the individual components. The maximum

crosstalk values (for 81X driver) which correspond to a trace on the top layer, have been used.

- Crosstalk Noise for Initial Breakout Pitch = 0.19V
- Crosstalk Noise for Intermediate Pitch = 0.2V
- Crosstalk Noise for Final Pitch = 0.17V

which gives a total crosstalk noise comes to 0.56V.

The reflection noise component comes to 0.03V and with an estimated SSN of 0.2V, the total noise can be obtained by computing the root sum squares of the individual components. The total noise is therefore,

$$V_{n,total} = \sqrt{V_{crosstalk}^2 + V_{reflection}^2 + V_{SSN}^2} = 0.6V \tag{4.3}$$

The total noise is well within the noise budget of 0.7V. The worst case off-chip skew on an 8cmx8cm substrate is around 0.2ns. After adding estimated on-chip skew component of 100ps and a jitter component of another 0.2ns, we can expect a cycle time of at least 2ns. The total I/O bandwidth is then

$$I/O\ Bandwidth = 2000\ x\ 500\ x\ 10^6 > 100GB/s \tag{4.4}$$

CHAPTER 5

A Review of Address Lookup Approaches

This chapter surveys existing address lookup algorithms and classifies them based on the approach taken. To reduce the size of the routing tables, routers store variable length prefixes, each of which corresponds to a network. To forward a packet, a router needs to find the longest matching prefix for the destination address. For instance, Table 5.1 shows entries of a sample routing table. For a destination address whose first 8 bits are 01110101, multiple prefixes in the table have a match. Of those, the longest match is with entry 011101*, and the corresponding next hop address is therefore 6. Most lookup algorithms fall into two categories: *Trie* based algorithms and *Binary Search* based algorithms. Various other modifications have been suggested to improve the address lookup speed and these are discussed toward the end of the chapter. Table 5.4 [89] at the end of the chapter summarizes the complexity of various lookup algorithms.

5.1 Trie Based Schemes

Trie based schemes use the “thumb indexing” method of dictionaries [75]. Prefixes are stored as paths from the root node to the leaf node. Searching for the longest matching prefix requires checking a few address bits at a time which directs search to a certain

Table 5.1: A sample routing table with prefixes and next hops

Prefix	Next Hop
0010*	2
10*	9
01*	3
0111*	5
1011*	1
011101*	1
11001*	6

portion of the sub-trie. In a binary implementation, each bit in the address is checked and a bit 0 points to the left half of the trie and a bit 1 points to the right half of the trie. The trie is traversed till a leaf node is reached which determines the longest matching prefix. In the worst case, the number of memory accesses required for these schemes to determine the longest matching prefix equals the depth, D given by:

$$D = (Address\ Bits)/log_2(M)$$

where M is the degree of the trie. Most trie based schemes attempt to reduce the number of memory accesses by reducing the trie depth.

5.1.1 Binary Trie Schemes

Various schemes have been proposed to reduce the average depth of the trie. The earliest of these is the Patricia scheme [75], which reduces the depth of the trie by removing all internal nodes with only one child. This technique, which is also known as Path Compression, reduces long one-way branches. The idea behind this scheme is that if a search comes to a node with a long one-way branch, then there is only one possible outcome and a bit-by-bit traversal is not necessary. Instead, a count of the number of bits to be skipped in each branch of the trie can be maintained at each node. Now when a

search leads to a leaf node, the address is compared with the route information stored at the node. If there is a match, then the longest matching prefix is found; otherwise the search continues by backtracking to the parent node. The recursive backtracking makes the worst case search time D^2, where D is the depth of the trie as given before [76]. One way to limit the amount of backtracking is to cache pointers to intermediate nodes [77]. In this approach prefixes are broken down into 4 categories depending on their lengths and stored in separate CAMs. This limits the search range and keeps the worst-case node traversal time constant. The NetBSD implementation of the address lookup scheme [78] uses a variation of the Patricia algorithm which avoids recursive backtracking by maintaining pointers to parent nodes. The expected length of the search is $1.44log(Number\ of\ Entries)$, or about 22-23 memory accesses for a routing table with $>$30,000 entries. The approach by Doeringer et al. [79] with *dynamic prefix tries* uses similar techniques to avoid backtracking and allows easier key deletions. The worst case lookup time in these schemes reduces to $2D$. Filippi et al. [80] modified the DP-tries further to avoid backtracking completely. This is done by making additional comparisons in the forward phase and storing the best-forwarding decision at each step. By doing this, the worst case bound is further reduced to D.

Another technique to reduce the average depth of the trie is Level Compression (LC) [81, 82], which compresses parts of the trie that are densely populated. The idea in this scheme is to replace each complete sub-trie of height h with a sub-trie of height 1 with 2^h children. The binary trie, path compressed trie and the LC compressed trie representations for the prefixes in Table 5.2 are shown in Figure 5.1.

5.1.2 Multi-way Trie Schemes

The depth of the trie can also be reduced by using higher order tries. In a basic non-binary approach, the address bits are partitioned into sections of m bits each and each section is used to address a different level of M ($= 2^m$)-ary trie. The degree can be increased until all

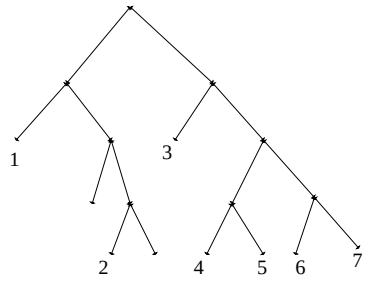
Table 5.2: Sample Prefix set

Number	Prefix
1	00*
2	0110*
3	10*
4	1100*
5	1101*
6	1110*
7	1111*

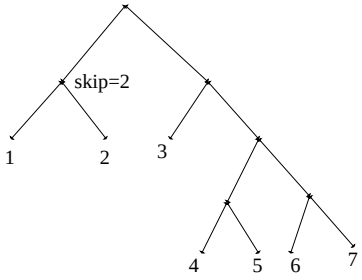
address bits are used in determining the longest matching prefix. The trie would then be very inefficient in terms of memory usage and the whole circuit would fall under the class of Content-Addressable Memories (CAMs). The time and space issues for tries of different degrees have been studied extensively in [83]. Until recently, CAMs were considered impractical for large routing tables due to their size and speed. Recently announced T-CAM (Ternary CAM) products (like the 32K x 144 NL877313 device from Netlogic [84]) run at about 80MHz which is comparable to DRAM speeds. One of the disadvantages is that for a core router application, multiple CAM chips are still required making the implementation expensive. However, it is expected that once densities and speeds for CAMs increase further, they would be an attractive option for route lookups.

One of the problems with going to a higher degree trie is that trie completion needs to be done to accommodate variable length prefixes. Also, some mechanism needs to be adopted to handle multiple next hop addresses on the same path. In one method, called *leaf pushing* [85], only the leaves of the tree are allowed to store next hop information. Another approach by Filippi et al. [80] stores additional flags that indicate whether a next hop address or a pointer is stored at a node. This avoids memory consumption in adding extra nodes.

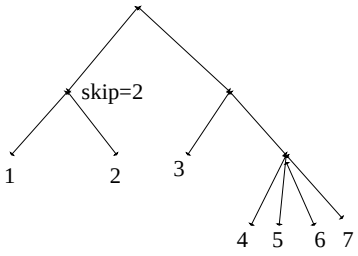
The degree of the trie need not be constant through all levels of the trie. For example, schemes in [86, 87] suggest using an initial 16-bit array lookup before using other scheme



(a) Binary Trie



(b) Path-compressed Trie



(c) LC trie

Figure 5.1: Binary, Path Compressed and LC compressed Representations

to determine the longest matching prefixes. This is guided by the fact that for most IPv4 routing tables, there are very few prefixes of length less than 16. Most searches would not result in a hit till after 16 address bits have been traversed and reducing the memory accesses till this stage to a single lookup leads to a good performance improvement. The scheme in [88] improves upon the two stage lookup in [86] by using variable-sized second stage arrays. Multibit tries [85] and multiresolution tries [89] both explore variable length degree tries. These algorithms perform much better than binary trie implementations as the number of memory accesses becomes much less (less than half in the worst case [89]).

5.1.3 Compressed Tries

A number of approaches have focused on compressing the trie information to reduce the size of the forwarding tables. In one approach [90], Degermark et al. break the trie

structure into 3 levels, each of which could require up to 4 memory accesses. The information at each level, i.e. whether a node continues beyond this level or ends at the current level or before, is coded as a code word array along with a base index array and a maptable. Using this scheme, a routing table with 40,000 entries can be compacted into 150-160 KB, which is comparable to the size of L2 caches in current CPU designs. Another way to compress the trie information has been suggested by Tzeng et al. [89, 91]. In their approach, instead of storing collective data for each node, a unique number is assigned to each trie with a different topology. For instance, a prefix node with one left child could be coded as 110; with two children as 111. By representing the entire trie with these unique codes, a significant compression can be achieved. The memory required to support a 128K-entry route table is about 3MB. Huang et al. [88] use a compression technique to reduce the size of the next-hop array by marking the beginning of each stage of a new set of next hops with a 1 in the compressed bit map array. The overall memory requirement for this scheme is in the range of 1.5MB-2MB for a routing table with around 40K entries. Another way to reduce the memory consumption, suggested by Uga et al. [77] is to eliminate common parts of addresses of adjacent nodes. This reduces the memory size by about one-third.

5.2 Binary Search Based Schemes

In binary search based schemes, the value of the entry is checked against the median value of each subtree. If the value is less than the median value, the search is directed to the left half of the subtree and if it is larger, to the right half. The number of memory accesses for an X-way search over N entries is:

$$M = log_X N$$

Binary search by itself does not work for longest matching prefixes. This is due to the

Table 5.3: Modified Binary Search Table from [1]

		$>$	$=$
P1)	100000	P1	P1
P2)	101000	P2	P2
P3)	101010	P3	P3
	101011	P2	P3
	101111	P1	P2
	111111	-	P1

fact that binary search works for numbers whereas prefixes store ranges. Various schemes have been proposed to adapt binary search for the longest-matching prefix problem. Lampson et al. [1] suggest encoding each prefix as a range of values and then searching on the number of entries. For instance, a prefix of 1* actually represents a range from 100000 to 111111 (assuming 6-bit values). Therefore, each entry is represented by two numbers. The problem of finding the longest matching prefix is translated to the problem of finding the narrowest enclosing range. As an example, the three prefixes 1*, 101* and 10101* would be represented by Table 5.3 where the $>$ and $=$ pointers direct the search towards the narrowest range [1]. Since the number of entries gets doubled, search takes O($log_2(2N)$) time. A multi-way search can be done to improve the complexity. The authors also suggested a good way to implement binary search for addresses larger than the machine word size. Their measurements indicate a lookup time of 2-3 million lookups/sec.

Waldvogel et al. [87] proposed a hash based scheme where a search is performed on the number of possible prefix lengths. The complexity of their scheme is $log_2 W$ hash computations, where W is the length of the address in bits. Instead of a linear search through decreasing prefix lengths, they start at the median length and progress using a binary search. To solve the problem of directing search to the correct sub-tree, they store additional markers in the hash tables. Measurements provided by the authors suggest a performance capability of 5-10 million lookups/sec.

Yazdani et al. [92] have also suggested binary and multi-way searches on prefixes. They handle prefix ranges by building enclosures of data sets. The enclosure is then treated as a single point. This approach however, leads to a variable number of memory accesses to determine the next hop.

5.3 Other Schemes to Improve Lookup Performance

Various other schemes have been suggested to improve the performance of lookups. In the label swapping technique, a label is used to expedite packet identification [93–95]. These schemes are useful for small networks but are not very scalable. They also require a label binding protocol which takes some amount of time.

Chiueh et al. [96, 97] have designed caches specifically for network processors. By using address ranges instead of addresses to cache entries, performance of caches can be improved considerably over caches in general purpose processors. However, caches rely on the temporal locality of data which is not necessarily true for core routers.

Another approach to speeding up the lookup is to use an array of processors. Knox et al. [98] use a linear pipelined array of processors to implement the Patricia algorithm. To traverse the trie, each processor is assigned to a level of the trie so the number of processors depends linearly on the number of address bits.

Cheung et al. [99] in their paper describe how to optimally construct routing tables. Given a memory hierarchy (size, speed) they find the optimal generalized level-compression transformation that needs to be performed such that the average lookup time is minimized.

Table 5.4: Complexity of Route Lookup Algorithms (W is the number of address bits, N is the number of prefixes and k is an algorithm dependent constant)

Algorithm	Insertion	Deletion	Longest Match
Patricia Trie			$O(W^2)$
Dynamic Prefix Trie	$O(W)$	$O(W)$	$O(W)$
Multiway and Multicolumn Search	$O(N)$	$O(N)$	$O(log_k(N)+1)$
Binary Search	$O(N)$	$O(N)$	$O(log(N)+W)$
Complete Prefix Tree Search	n/a	n/a	$O(W/k)$
Binary Hash Table Search	n/a	n/a	$O(log(W))$
Large Memory Architecture	n/a	n/a	$O(log(W/k))$
Level Compressed Trie	n/a	n/a	$O(log(W/k))$
Hashed Radix Trie	n/a	n/a	$O(W/k)$

CHAPTER 6

Description of Route Lookup Schemes

In this chapter we propose some route lookup schemes that perform better than existing comparable schemes. The first of these is a hardware, trie based scheme where the routing table information is compacted such that it is small enough to fit on an on-chip SRAM. Trie traversal then occurs only in the on-chip SRAM and finally a single DRAM lookup is required to determine the next hop address [100].

The other schemes adapt binary search to work for variable length prefixes by storing an additional field containing path information for each node. This results in a complexity of $O(log(N))$, where N is the number of entries, as opposed to $O(log(2N))$ of other binary schemes.

6.1 Description of the Trie-Based Scheme

This scheme was targeted to work for an Optical Burst Switching(OBS) environment [101]. The main goal of the scheme was to minimize off-chip accesses while maintaining a fast, constant lookup time. The main factors that motivated this work were:

1. Speed - To support Gigabit/Terabit routing in the future, speed is a crucial factor. To

achieve the required high lookup rates, a hardware implementation is the only alternative. The fastest software approaches still take hundreds of nanoseconds to perform a lookup, which is not acceptable.

2. Scalability - To accommodate larger numbers of hosts/addresses, routing table sizes are expected to increase in the future. This makes the route lookup problem harder because more memory accesses are required to determine the next hop address. Moreover, a large forwarding table cannot be stored in an on-chip memory or a cache and therefore, very expensive off-chip accesses need to be made.

3. Constant Lookup Rate - This requirement is specific to the Just-In-Time (JIT) signaling scheme used in OBS networks.In optical burst switched networks, signaling is done out of band, and only the signaling channel goes through an O/E/O conversion [102]. The signaling message is sent ahead of the data burst and is interpreted at each of the nodes along the path. The switches along the path are configured before the data burst arrives. The data burst is sent after a delay (without receiving any confirmation about available path) which depends on the number of hops along the path [103]. If the set-up time at each node is variable, it would make the delay more unpredictable and would lead to a more inefficient network.

The rest of the section discusses the details of the scheme and the accompanying hardware implementation.

6.1.1 Data Structure

The information in the routing table is split into an on-chip SRAM and an off-chip DRAM. The SRAM and the DRAM databases are constructed from the conventional multi-way trie structure. The data in the SRAM does not contain any “real” information like next hops or pointers. Instead, the offsets (equivalent to pointers) are calculated from the bit pattern in the SRAM. The levels in the trie are traversed by computing the offsets

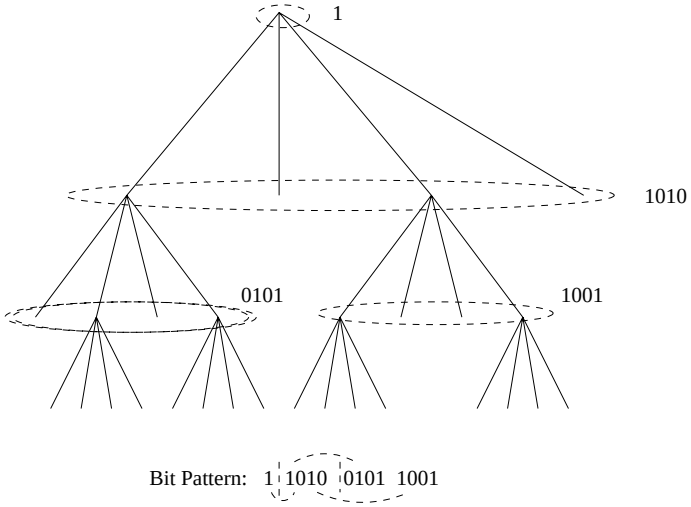


Figure 6.1: Sample 4-way trie and the corresponding bit pattern

for each level. This transfers the burden of finding offsets (or pointers) to hardware which is faster than reading data from memory. In addition, this reduces the amount of memory required because no pointers (or indices) need to be stored. The SRAM can be thought of as storing the path information of the trie while the corresponding DRAM stores the actual data associated with each node (i.e. the next-hop addresses).

The SRAM is built by writing a bit for every node with all its children in the trie structure. Each of the children in the node gives rise to a similar 1 or a 0 depending on the presence or absence of its child nodes. As an example, consider the 4-way trie shown in Figure 6.1. For this trie, the SRAM contains a “1” for every node with its 4 children. Each 1 in the SRAM bit-pattern gives rise to 4 more bits in the bit-pattern as shown in Figure 6.1. A “0” is not propagated while generating the bit-pattern. Also, each “1” in the SRAM corresponds to a row in the off-chip DRAM which stores the 4 possible output port numbers for each of the 4 children. In practice, a DRAM row would hold more next hops and determining the correct DRAM row and column is easy from the bit pattern. This leads to a very compact structure making the SRAM size much smaller than the corresponding DRAM trie structure.

The route-lookup is done in two stages. The first stage involves only SRAM lookups

and the longest path corresponding to the address is determined from the bit-pattern stored in the SRAM. At the end of this stage, the row and column address of the DRAM where the corresponding next-hop address is stored can be determined. In the next stage, a single DRAM lookup is done and the next-hop address is read. Only a single DRAM access is required in this implementation, reducing the total lookup time. The two stages can be pipelined to give a result every 60-65 ns (random access time for a DRAM) giving over 15 million lookups per second. To further improve speed, multiple DRAMS containing identical information can be used in parallel.

6.1.2 Building the data structure

The data structure to be stored in the SRAM is built from the corresponding binary trie. The pseudocode for building the SRAM and DRAM data is given in Algorithm 1.

Our implementation uses a 16-way trie, although any degree of trie can be built. The steps involved in the building the trie are as follows:

Step 1 Each entry from the routing table is read and stored in a list.

Step 2 The list is sorted in an ascending order. For prefixes of different lengths, the prefix with the smaller length is considered to be smaller. For example, 10* would be considered smaller that 100*. The reason for doing this is to ensure that a smaller prefix is always entered first in the trie structure. If the reverse were to happen, additional steps would be required to ensure that correct next-hop entries are stored in child nodes.

Step 3 The root node is created and each of the child node pointers are initialized to NULL.

Step 4 Each entry from the list is read and expanded if necessary to complete the trie. The trie is traversed and the child node pointers and next hop addresses are updated accordingly.

Algorithm 1 Building Procedure for SRAM and DRAM data

for $i = 1$ to N **do** {* N is the number of routing table entries *}
  $Array[i] \leftarrow$ Prefix Data
end for
Sort *Array*
{* Building the trie structure *}
for $i = 1$ to N **do** {* Take each entry and place it in its right place in the trie *}
  {* Start at the root node *}
  $current\_node \leftarrow root, current\_mask \leftarrow prefix\_mask$
  while $current\_mask > 1111...0$ **do** {* Traverse till the prefix length *}
    $ext\_bits \leftarrow$ Extracted address bits
    if $current\_node \rightarrow children[ext\_bits] ==$ NULL **then** {* Add new node to the trie *}
      $new_node \rightarrow children ==$NULL
      $new\_node \rightarrow next\_hop == current\_node \rightarrow children[ext\_bits]$
      $current\_node \rightarrow children[ext\_bits] = new\_node$
    else
      {* Traverse the trie further *}
      $current\_node \leftarrow current\_node \rightarrow children[ext\_bits]$
    end if
    Left shift *current_mask* by M bits (M=degree)
  end while
  Expand the last part for trie completion
  Update associated next hop entries
end for
{* Generating SRAM and DRAM data *}
ENQUEUE *root*
for Each node *current_node* in trie **do**
  if *current_node* = DEQUEUE is not NULL **then** {* Check to see if stack is not empty *}
    Check all children of *current_node* and ENQUEUE all existing children
    Generate M bit pattern (for SRAM) depending on the presence/absence of children
    Write corresponding *next_hops* in DRAM
  end if
end for

The depth of the trie is $\frac{(No.\ of\ Address\ bits)}{(log_2 M)}$, where M is the degree of the trie. An insertion of an entry into the trie structure can take up to these many lookups in the worst case. Since building the trie requires inserting N entries, where N is the total number of entries in the routing table, the cost of building the trie is $N * D$, where D is the depth of the trie.

Once the trie is built, the compact SRAM data structure can be constructed by doing a breadth-first traversal on the trie.

6.1.3 Searching the Data Structure

Algorithm 2 outlines the procedure for performing a search for a given address.

Algorithm 2 Search Procedure

```
current ← Start bit position of SRAM data
level ← 0
ext_bits ← First m bits of the address (2^m = M = Degree of Trie)
while level < Max_level do
  if SRAM[level][current + ext_bits] == 1 then {* Need to go further down the trie *}
    sum ← Sum of previous 1's in this level
    Increment level
    current ← SRAM[level][sum * M]
  else {* SRAM trie traversal is over *}
    dram_row ← Calculate all sum of 1's till current position
  end if
end while
```

The main steps in the algorithm are summarized below:

Step 1: The start pointer is initialized to the first M-bit pattern in the SRAM data structure, where M is the degree of the trie.

Step 2: The first $log_2(M)$ bits of the address are read. For a 4-way trie this would mean 2 bits and for a 16-way trie, 4 bits of the address.

Step 3: These address bits are used as the offset in the M bit pattern from the start pointer.

Table 6.1: A sample database of prefixes and their associated hops

Prefix	Prefix Length	Next Hop
128.0.0.0	2	3
128.0.0.0	4	6
140.0.0.0	8	3
140.12.0.0	16	2
64.0.0.0	2	7
64.0.0.0	8	12
38.0.0.0	8	5
112.0.0.0	4	9
112.48.0.0	14	5
80.0.0.0	4	2

Step 4a: If the bit indicated by the offset is 1, then the start pointer is moved to the next level. The position of the start pointer is calculated by computing the sum of all the previous 1's in the level and multiplying it by M. Steps 2-4 are repeated.

Step 4b: If the bit indicated by the offset is 0, then the search terminates. The total number of 1's before and including the parent 1 (the 1 that led to the 0) gives the DRAM row number containing the next hop address.

6.1.3.1 Example for a 16-way trie

Table 6.1 shows a sample set of prefixes along with their prefix lengths and the next hop addresses. The prefix length is the number of valid bits of the prefix in the routing table. The first step in building the prefix trie is to sort the prefixes in ascending order as described previously. The sorted prefixes are shown in Table 6.2.

To build the entire trie structure, the prefixes are added one at a time from the sorted list. Trie-completion is performed wherever necessary to ensure that only leaves represent valid prefixes. In the figures shown below, the digit beside each leaf trie represents the next hop address and a "*" represents the default next hop. Figure 6.2 shows the trie data structure at initialization, where all the leaves of the root node point to the default next hop

Table 6.2: Prefixes from Table 6.1 sorted in ascending order

Prefix	Prefix Length	Next Hop
38.0.0.0	8	5
64.0.0.0	2	7
64.0.0.0	8	12
80.0.0.0	4	2
112.0.0.0	4	9
112.48.0.0	14	5
128.0.0.0	2	3
128.0.0.0	4	6
140.0.0.0	8	3
140.12.0.0	16	2

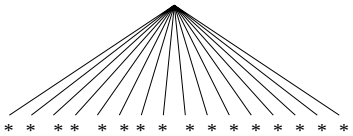


Figure 6.2: Trie Structure at initialization

address.

The first prefix from the sorted list (00100110*) is added to the trie structure. The first 4 bits of the prefix point to leaf 3 of the root node. Since the prefix is longer than 4 bits, a new node is added to leaf 3 of the root node. The new node inherits the next hop addresses of the parent node (in this case, *). The next 4 bits of the prefix point to leaf 6 of the new node. As a result, the next hop address of leaf 6 is changed to 5 (the next hop address of the prefix), as shown in Figure 6.3. In this way, any address matching less than 00100110* is pointed to the default next hop address whereas a complete match will get a next hop address of 5.

The next entry in Table 6.2 to be added to the trie structure is 01* with a next hop of 7. Since the prefix length is not a multiple of 4, trie-completion needs to be done. The range of prefixes represented by 01* are 0100*, 0101*, 0110* and 0111*. Accordingly, next hop addresses associated with leaf 4, 5, 6 and 7 of the root node are changed to 7. This is

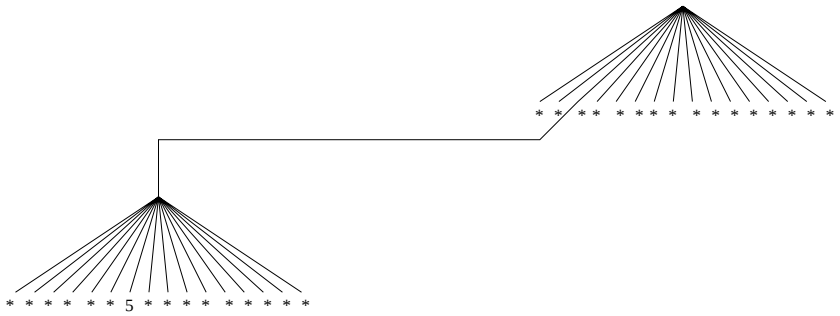


Figure 6.3: Trie Structure after adding prefix 00100110* to Figure 6.2

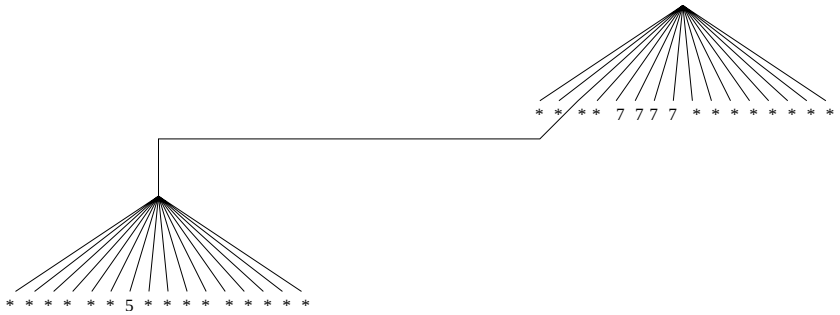


Figure 6.4: Trie Structure after adding prefix 01* to Figure 6.3

shown in Figure 6.4.

The next entry to be added to the trie structure is 01000000* with a corresponding next hop address of 12. A new node is added to child 4 of the root node, corresponding to the first 4 bits of the prefix (0100). Children of the new node inherit the next hop address of 7 from the parent node. The first child of the new node gets the next hop address of 12, since the next 4 bits of the prefix point to the first child, as shown in Figure 6.5.

The next entry of 0101* results in the next hop address of child 5 of the root node to

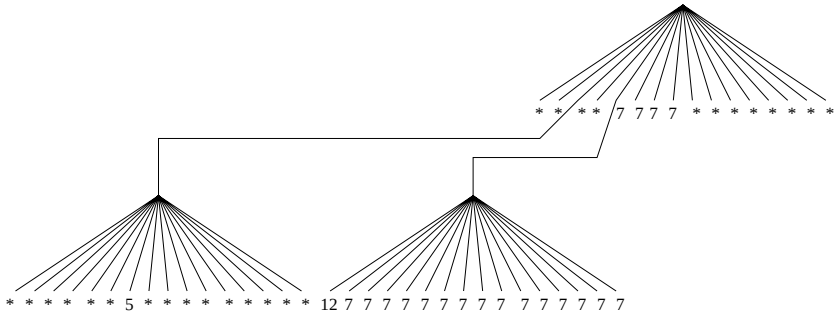


Figure 6.5: Trie Structure after adding prefix 01000000* to Figure 6.4

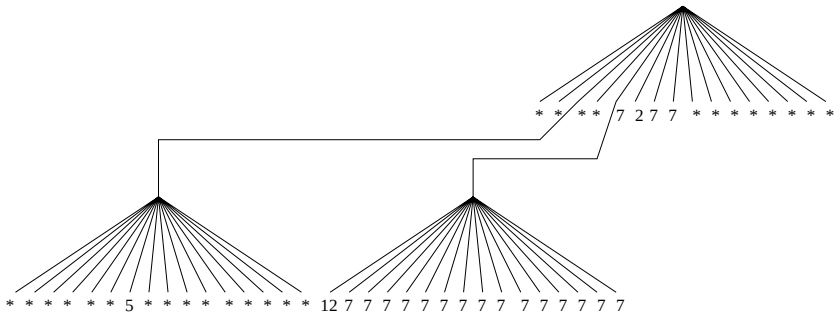


Figure 6.6: Trie Structure after adding prefix 0101* to Figure 6.5

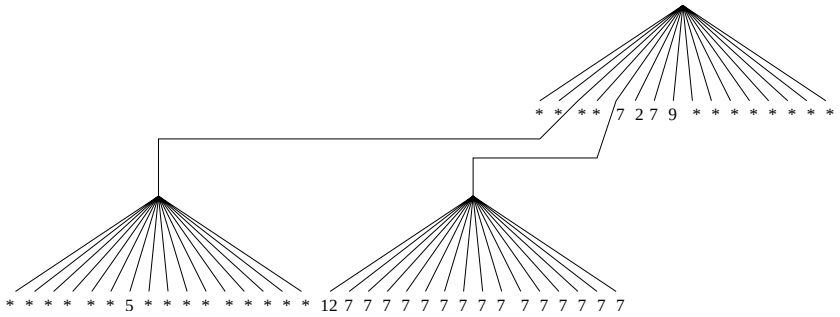


Figure 6.7: Trie Structure after adding prefix 0111* to Figure 6.6

be changed from 7 to 2 as shown in Figure 6.6.

The next prefix to be added is 0111* with a next hop of 9. Figure 6.7 shows the result of adding this entry to the trie structure. The next hop address of child 7 of the root node is changed from 7 to 9.

Figure 6.8 shows the trie structure after the entry 01110000001100*, with an associated next hop of 5, is added. Two additional nodes are added to the trie structure and finally trie completion is required.

The next entry to be added is 10* with a next hop address of 3. Figure 6.9 shows the trie structure after the addition of this entry. In particular, the next hop addresses associated with children 1000, 1001, 1010, 1011 of the root node, are changed from the default (*) to 3.

The next entry of 1000* results in child 8 of the root node getting a next hop of 6 as shown in Figure 6.10. This also illustrates the importance of placing 10* before 1000*. Had the reverse been done, the next hops of the children of root node would have been

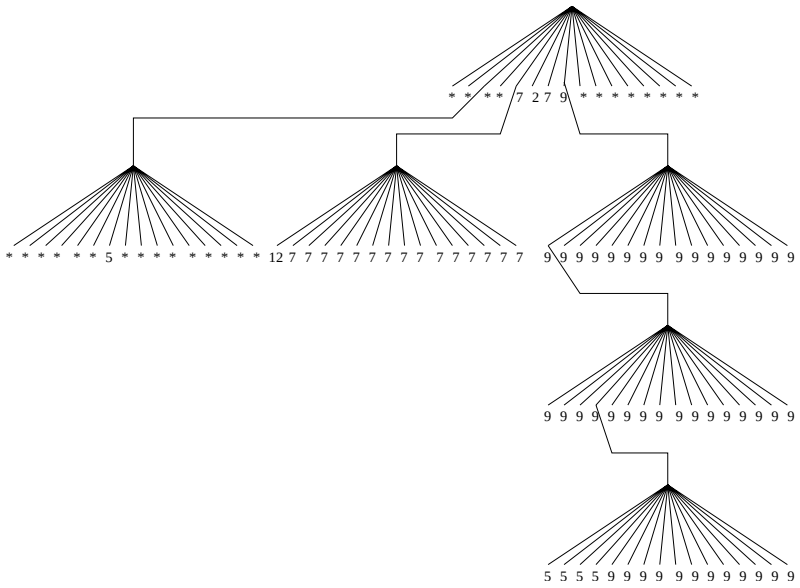


Figure 6.8: Trie Structure after adding prefix 01110000001100* to Figure 6.7

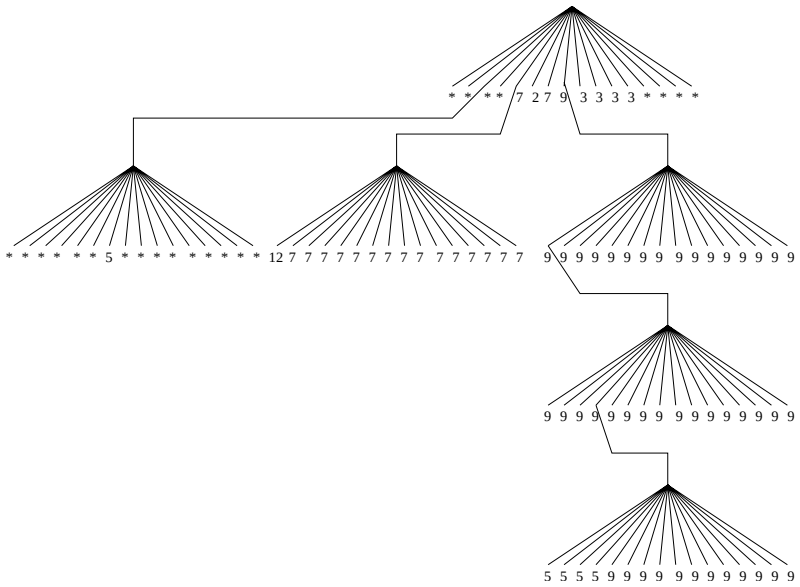


Figure 6.9: Trie Structure after adding prefix 10* to Figure 6.8

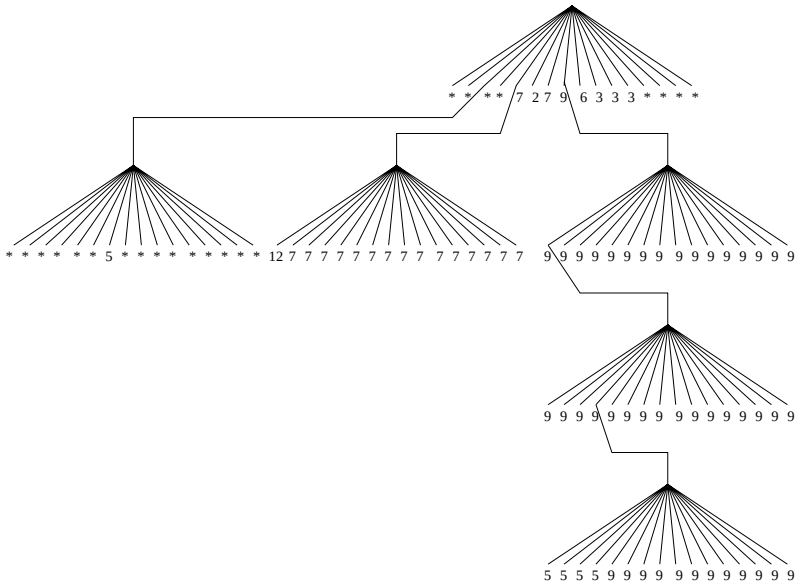


Figure 6.10: Trie Structure after adding prefix 1000* to Figure 6.9

incorrectly set to ****72793333****.

Figure 6.11 and Figure 6.12 similarly show the trie structure after the addition of the last two entries of 10001100* and 1000110000001100*.

The bit pattern that the SRAM stores for each of the levels is shown in Figure 6.13. The first few bits of each SRAM row (the “Sum” column) contain the sum of 1’s in the current level in previous rows. This is useful to maintain for the following reason. If the SRAM bit-pattern for a particular level spans more than one SRAM row, multiple SRAM accesses would be required to compute the sum of 1’s. By adding additional bits in each SRAM row to store this sum, only one SRAM access per level is required. This sum value is easy to compute while generating the SRAM bit-pattern and adds a small overhead to the memory consumption. In this case, since the bit pattern of each level fits in a single SRAM row, these bits are all 0. The off-chip DRAM needs to store only the next hop addresses as shown in Figure 6.14, where the * represents the default next hop. Each row in the DRAM shown in the figure contains 16 (equal to the degree of the trie) next hop entries. This is only a logical organization and in practice two or more of these rows can be merged together. The row and column address of the DRAM would still be determined

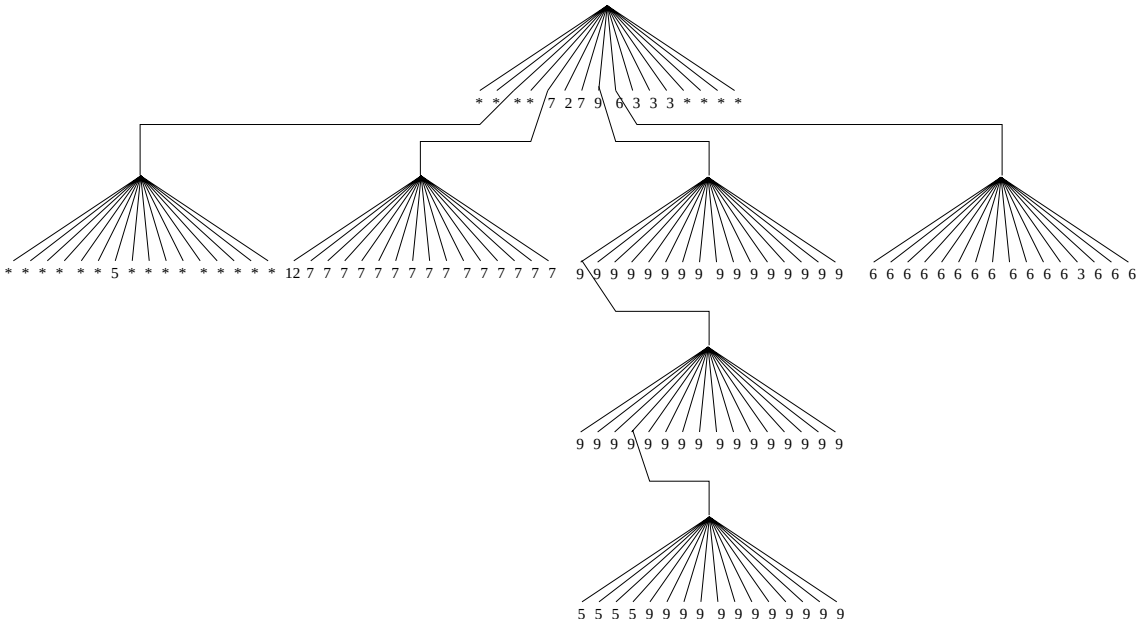


Figure 6.11: Trie Structure after adding prefix 10001100* to Figure 6.10

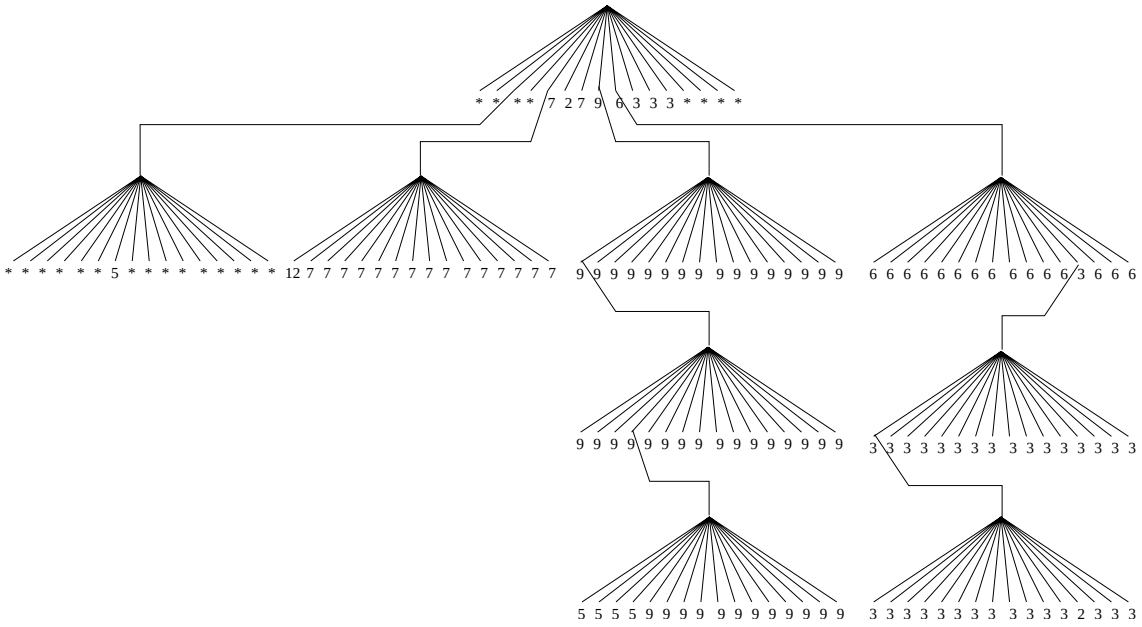


Figure 6.12: Trie Structure after adding prefix 1000110000001100* to Figure 6.11

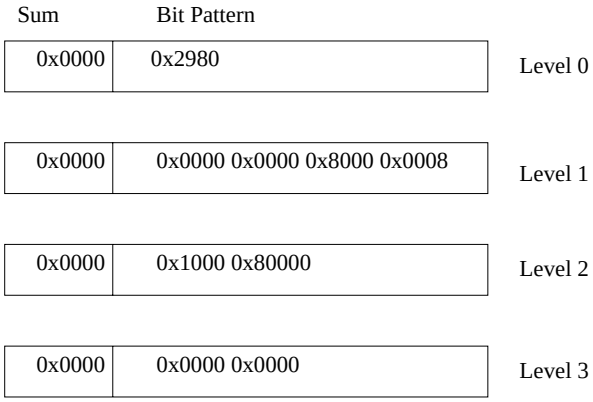


Figure 6.13: Bit pattern of the trie as stored in the SRAM

```
* * * * 7 2 7 9 6 3 3 3 * * * * Row 0
* * * * * * 5 * * * * * * * * * Row 1
12 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 Row 2
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 Row 3
6 6 6 6 6 6 6 6 6 6 6 3 6 6 6 Row 4
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 Row 5
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 Row 6
5 5 5 5 9 9 9 9 9 9 9 9 9 9 9 9 Row 7
3 3 3 3 3 3 3 3 3 3 3 3 2 3 3 3 Row 8
```

Figure 6.14: Data stored in the DRAM

easily while traversing the SRAM.

As an example, suppose that the longest prefix match of an incoming IP address of 112.48.32.248 needs to be determined. The first 4 bits of the address (0111) lead to bit 7 (starting from bit 0) in the bit pattern for level 0. This bit is set to 1 in the SRAM row and so the search has to continue to the next level. There are two 1's in the bit pattern before bit 7 in this level as seen from Figure 6.15. This means that for the next level (level 1), the first 2*16 bits have to be skipped. This leads to the starting bit of the pattern 0x8000 in level 1.

The next 4 bits of the address are 0000 which lead to bit 0 in the 0x8000 pattern (See Figure 6.16). This is also set to 1 so the whole process is repeated. There are no 1's before

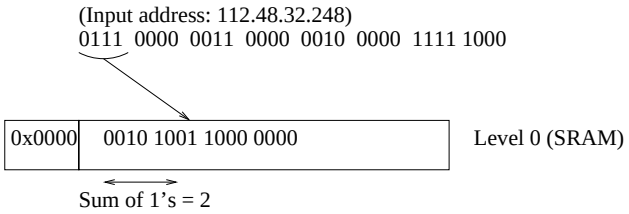


Figure 6.15: Searching Level 0 in SRAM

this bit in level 1, so none of the bits need to be skipped in the next level (level 2).

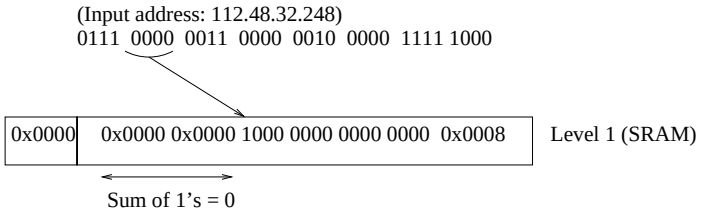


Figure 6.16: Searching Level 1 in SRAM

The next 4 bits of the address are 0011 which point to the bit 3 in the pattern 0x1000 (Figure 6.17). This bit is set to 1 and the process is repeated once again. As before, there are no bits to be skipped for the next level.

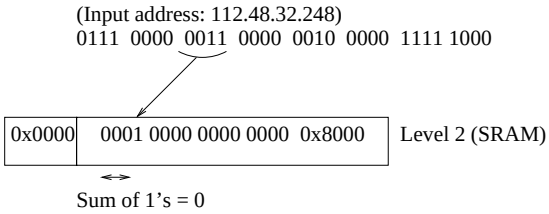


Figure 6.17: Searching Level 2 in SRAM

The next 4 bits of the address are 0000 which points to bit 0 in the bit pattern 0x0000 of level 3. Now this bit is set to 0 as seen from Figure 6.18, which means that there is no possible longer prefix. The total number of 1's before and including the 1 in level 2 are 7. This gives the row number of the DRAM where the next hop addresses are stored. The column address can be determined from the number of bits used to store a hop address and by using the next 4 bits from the address as the offset. The last 4 bits in the address pattern

(0000) which gives an offset of 0. The next hop address is therefore 5 as seen from Figure 6.14.

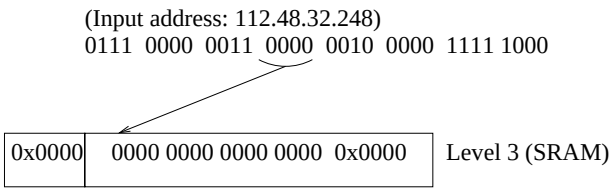


Figure 6.18: Searching Level 3 in SRAM

6.1.4 Insertion and Deletion

Any updates to the routes that change only the next hop address, require only the DRAM data to be modified. The SRAM bit pattern does not change in this case. However, any insertion or deletion of an entry that results in the addition or deletion of a node, changes the SRAM bit pattern. There is no efficient way to modify the SRAM data other than to build the structure from scratch from the trie. This is not a problem, because updates to the routing table do not occur as frequently as searches and multiple updates can be batched to improve the performance. This is common practice in other algorithms as well [1, 87, 90]. Building the SRAM data takes on the order of a 100ms on a Sun Ultra 5 with a 333 MHz processor. Most forwarding tables require to be updated once every second, which can be easily accomplished.

6.1.5 Hardware Implementation

Figure 6.19 shows the block diagram of a generic router. The trend in recent router designs has been to push more intelligence into the line cards to increase message handling capacity [104]. Each of the line cards contains a copy of the forwarding table generated from the main routing table. The forwarding tables in the line cards get updated every few seconds. Operations like address lookup, scheduling and configuring the switch

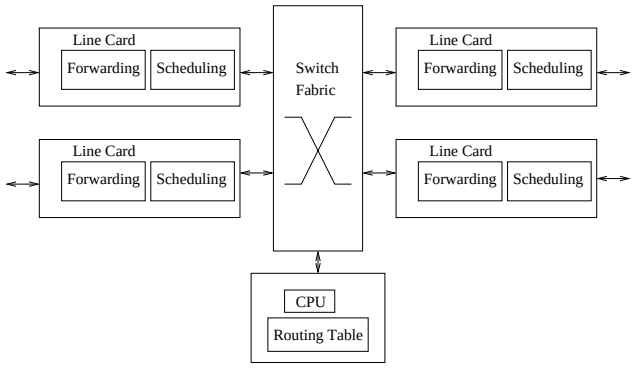


Figure 6.19: Block diagram of Router

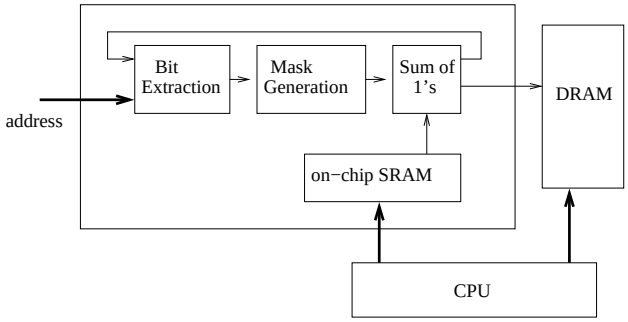


Figure 6.20: Block diagram of the Forwarding Engine

fabric are performed by the line cards themselves.

The most time critical part in the design of the router is the forwarding, i.e. determining the next hop address from the packet destination address. The block diagram of the forwarding engine is shown in Figure 6.20. In our implementation, a 16-way trie is used to build the data structure. The bit extractor therefore picks the next 4 bits of the address. This, along with the offset, is used to generate a mask for computing the sum of 1s. The sum of 1s unit takes the mask and the SRAM row to determine the next offset. Once the SRAM traversal is complete, a read request for the off-chip DRAM is generated and after the DRAM access time, the next hop address is available. The data stored in the SRAM and DRAM is generated in software.

The SRAM traversal is implemented as a Finite State Machine (FSM) and Figure 6.21 shows the state diagram for traversing the bit-pattern in the SRAM. Each state takes 8ns to

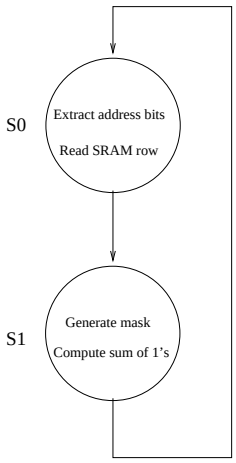


Figure 6.21: State Diagram for traversing SRAM

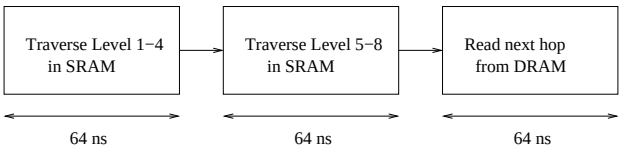


Figure 6.22: Pipeline stages of the forwarding engine

complete and so a total of 16ns is taken to traverse one level in the SRAM bit-pattern. Since there are a total of 8 levels to be traversed in the SRAM (because of 32-bit addresses in IPv4), it would take 128ns to traverse the SRAM. The loop in Figure 6.21 can be unrolled and pipelined more than once to increase the throughput. In our implementation, we unrolled the loop once and pipelined the two FSMs, to give results every 64ns as shown in Figure 6.22. This was done to match the DDR DRAM random access time [57]. The main hardware block used in the design of the forwarding engine is the unit that computes the sum of 1s till a given bit position in the SRAM row. The design for this is discussed next.

6.1.5.1 Generating the Mask

To compute the sum of 1s till a certain bit position, we generate a mask to remove the unwanted bits from the SRAM row. For example, consider the SRAM row in Figure 6.15.

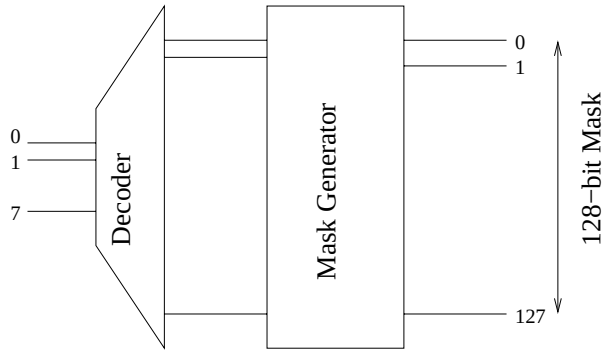


Figure 6.23: Generation of Mask from the bit position

The sum of 1s needs to be computed till bit 7 so the corresponding mask that is generated is

1111 1110 0000 0000

This is bit-wise ANDed with the SRAM row to give

(1111 1110 0000 0000) & (0010 1001 1000 0000) =

(0010 1000 0000 0000)

The result obtained after bit wise ANDing the mask with the SRAM row is given as the input to the unit computing the sum of 1's.

To generate the mask, the bit position is first decoded and depending on the 8-bit input, one of the output bits of the decoder goes high. The output of the decoder feeds into the mask generator circuit as shown in Figure 6.23. The mask generator is a very simple circuit where the inputs are connected to the outputs as shown in Figure 6.24. The delay through the generator is the maximum delay at line 127 with a fanout of 128. The 8:128 bit decoder takes around 0.7ns (in 0.18μ technology) to decode [105] while the mask generator again takes around 0.6ns to complete (see Appendix).

6.1.5.2 Computing the sum of 1s

Sum of 1s can be computed in a number of ways. The simplest way is to use a bank of adders. For a 128-bit wide SRAM row, the adders that would be required is shown in

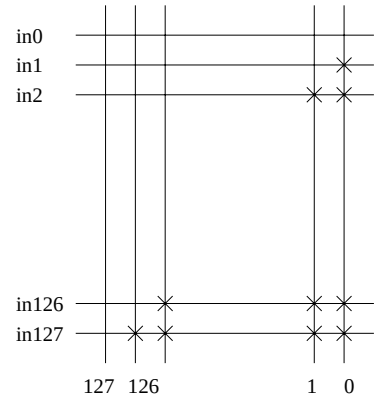


Figure 6.24: Mask Generator circuit

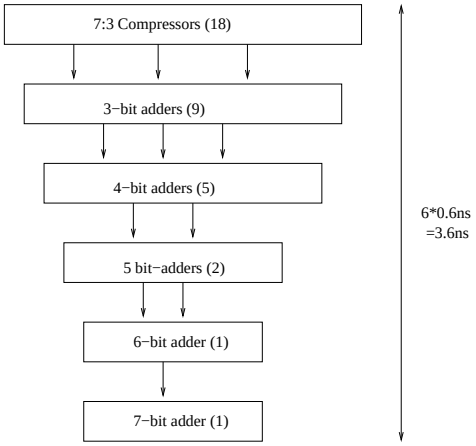


Figure 6.25: Adders used in the computation of the sum of 1's

Figure 6.25. The 7:3 compressors used in the first row add up 7 1-bit numbers and reduce the result to a 3-bit number. A 7:3 compressor can also be thought of as a counter that counts the number of 1s in the input. A 128-bit input requires 18 of these compressors. The 18 3-bit results obtained after this stage can be added in the next stage using 9 3-bit adders to give 9 4-bit results. This process is repeated till we get a final 8-bit result. The total number of adder stages used is 6 as shown in Figure 6.25. For a 0.18μ technology, a 32-bit adder takes about 0.6ns [106]. We have kept the same budget for our smaller adders, even though smaller adders take less time. The total worst-case time taken to compute the sum would be less than 4ns. In all, the total time taken to compute the sum of 1s is well under the budget time of 8ns (for each state in Figure 6.21).

6.2 Description of the Binary Schemes

This section discusses route lookup schemes based on binary searching. The advantage of a binary scheme is that these schemes work very well for long addresses (as in IPv6). The complexity of the scheme is $O(log(N))$ where N is the number of entries in the routing table. Unlike trie-based schemes, the complexity does not depend on the size of the address. This makes binary schemes especially attractive for 128-bit sized IPv6. This section discusses in detail the algorithms and the modifications required to adapt them for variable length prefixes.

6.2.1 Description of the Algorithm

To understand the algorithm, we first look at a variable-degree tree constructed from the prefixes in Table 1.1 as shown in Figure 6.26. To place prefixes in their relative positions in the tree, two conditions were used. For two prefixes, $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$,

1. If $A \subset B$, then A is a parent of B (where the parent could be any node along the path from the node to the root of the tree)

2. If $A \not\subset B$ *and* $A < B$, then A lies on the left subtree of B. To compare A and B, if the prefix lengths of A and B are equal, i.e. $n = m$, then the prefixes can be compared by taking their numerical values. However, if $n \neq m$, then the longer prefix is chopped to the length of the shorter prefix and the numerical values compared.

By applying these conditions to all the prefixes, the tree in Figure 6.26 can be constructed.

The problem with performing a binary search on variable length prefixes can now be seen. By simply sorting the prefixes in some fashion and performing a binary search, it would not be possible to determine the longest matching prefixes. For instance consider a sorting defined as follows:

For the two prefixes $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$,

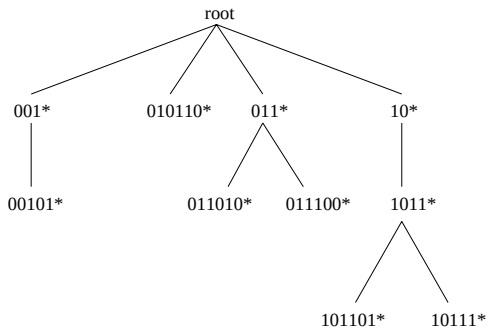


Figure 6.26: Binary Tree Constructed from Table 1.1

Table 6.3: Prefixes from Table 1.1 after sorting

Prefix	Next Hop
00101*	7
001*	4
010110*	5
011010*	6
011100*	1
011*	8
101101*	2
10111*	8
1011*	9
10*	3

1. If the prefixes are of equal lengths i.e. if $n = m$, then the numerical value of the prefixes determines which is the larger of the two.

2. If the prefixes are of unequal lengths i.e. if $n \neq m$, then the longer prefix is chopped to the length of the shorter prefix and the numerical values are compared. If after chopping the prefixes are equal, then the shorter prefix is considered to be the larger of the two.

Using the above definition, the prefixes in Table 1.1 can be sorted in ascending order as shown in Table 6.3. This is equivalent to having performed a post-order depth-first-search on the binary tree in Figure 6.26.

Performing a simple binary search for a given address on the sorted prefixes, would not necessarily lead to the longest matching prefix. For instance, prefixes 10110* and 100* both lie between the entries 011* and 101101* but both of them have different longest matching prefixes. In general, the prefix could be any of the parent nodes or the root (default) node. Therefore, to determine the correct next hop, additional (and a variable number of) steps would need to be performed. This problem arose because the node 101101* did not carry any additional information about its parent nodes. We can avoid this problem by storing an additional field at all nodes that gives information about all the parent nodes. This additional field, which we call the Path Information Field, is a 32 bit entry (for IPv4) where a 1 in any bit position in the field means that there is a parent node with a prefix till that bit position. For example, for the leaf node of 101101* the Path Information field would look like 0...101010, i.e. the second bit (corresponding to 10*), the fourth bit (corresponding to 1011*) and the sixth bit (corresponding to the leaf node itself) are set to 1. In addition, the node would also contain a pointer to a list of next hop addresses for the corresponding bits in the path information field. In this case, the list of next hop addresses would be 2,9,3. Now by looking at the path information field the longest matching prefix can be determined and the correct next hop address obtained from the list. The data structure used at the nodes is shown in Figure 6.27. In our implementation, we store the following information at each of the leaf nodes: the prefix, prefix mask, the next hop address corresponding to the leaf node, the number of internal nodes in the path and a pointer to the list of next hop addresses corresponding to the internal nodes. Table 6.4 shows the relevant information stored with each of the entries in Table 6.3. Using the same example, if the address 10110* were to be searched for in the list, the search would point to between the upper and lower entries of 011* and 101101*, respectively. A match between the address 10110* and the lower entry 101101* results in a match of up to 5 bits. By looking at the path information field, bit 5 is not set to 1. The next lower bit that is set, is bit 4. Therefore, the longest matching prefix for the given

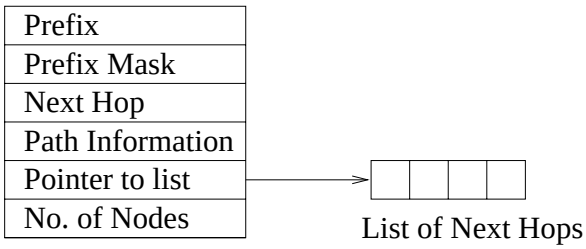


Figure 6.27: Data Structure used at a Node

address is 1011* and the corresponding next hop address in the next hop list is 9. In this case, the address needs to be compared with only the lower entry resulting from the search failure. To see why this is true, we look at different cases a search can end up in:

1. Between two leaf nodes with a common parent node. In this case either of the nodes can be used to determine the next hop address.

2. Between a node and its parent node. In this case, the parent node is the longest matching prefix and the parent node is the lower (larger) entry.

3. Between a node connected to the root node and a leaf node of the next branch (as in 011* and 101101*). In this case, nothing is gained by comparing with the smaller entry, since it can only lead to the root node (default next hop).

In all possible cases, it suffices to compare the address against only the lower entry to look for partial matches.

6.2.1.1 Building the Data Structure

The pseudocode for building the searchable data structure is given in Algorithm 3. The C code for binary schemes in this chapter is listed in the Appendices.

Building the data structure is fairly simple and the main steps involved are listed below:

Step 1 Each entry is read from the routing table and stored in an array.

Table 6.4: Search space of the prefixes in Table 6.3

Prefix	Next Hop	Path Information	Next Hop List
00101*	7	0...010100	4
001*	4	0...000100	-
010110*	5	0...100000	-
011010*	6	0...100100	8
011100*	1	0...100100	8
011*	8	0...000100	-
101101*	2	0...101010	9,3
10111*	8	0...011010	9,3
1011*	9	0...001010	3
10*	3	0...000010	-

Algorithm 3 Building Procedure (All Nodes)

{* *Array* refers to the initial array, $Array\_Leaf$ to the final *}
for $i = 1$ to N **do** {* N is the number of routing table entries *}
  $Array[i] \leftarrow$ Prefix Data
end for
Sort *Array*
$leaf\_ctr \leftarrow 0$
for $i = 1$ to N **do**
  if $Array[i].prefix$ is subset of $Array\_Leaf[leaf\_ctr].prefix$ **then** {* Insert in node list of $Array\_Leaf$ entry *}
    k $\leftarrow 0$
    while $Array[i].prefix$ is subset of $Array\_Leaf[leaf\_ctr-k].prefix$ **do** {* Check for all previous $Array\_Leaf$ entries too *}
      Update Path Information field of $Array\_Leaf[leaf\_ctr-k]$
      Increment the number of nodes field of $Array\_Leaf[leaf\_ctr-k]$
      Update $next\_hop$ list of $Array\_Leaf[leaf\_ctr-k]$
      Increment k
    end while
  end if
  {* Add a new leaf to the $Array\_Leaf$ *}
  Increment $leaf\_ctr$
  Generate $Array\_Leaf[leaf\_ctr]$ field entries from $Array[i]$ field entries
end for

Step 2 The entries are then sorted from the smallest to the highest. To compare prefixes, the rules listed previously are used. If prefixes are of equal lengths then the numerical values of the prefixes are used to compare them. If prefixes are of unequal lengths, then the longer prefix is chopped to the length of the smaller prefix and then compared. If after chopping, the prefixes are equal then the shorter prefix is considered to be the larger of the two. Doing this ensures that in the following step, the child nodes get processed before the parent nodes.

Step 3 Entries from the sorted list are then processed and added in an array, one at a time. Each entry is also tested with the last array entry to see if it is a subset of the array entry.

Step 3a If it is a subset, then the next hop information corresponding to the array entry is added to the next hop list of the array entry. The path information field of the array entry is updated and so is the field containing the number of nodes. This step is then repeated for previous entries till the test for subset fails. To see why this is necessary, consider the last prefix in Table 6.3. When prefix 10* is compared with the last array entry (1011*), it is added in the next hop list of 1011*. However, 10* also lies in the path of 10111* and 101101* and the corresponding entries need to be updated as well.

6.2.1.2 Searching the Data Structure

To search for the longest matching prefix, a binary search is performed on the entries. Algorithm 4 gives the procedure for obtaining the next hop address for a given prefix.

The search algorithm is summarized below:

Step 1 Binary search of the address is performed on the array entries which leads to a value between two consecutive array entries.

Step 2 The address is then matched with the lower entry, and checked against the path

Algorithm 4 Search Procedure (All Nodes)

```
{* Binary Search Part *}
start ← 1, end ← No_Leaves
while end > start + 1 do
   mid ← (start + end)/2
   if prefix < Array_Leaf[mid].prefix then
      end ← mid
   else
      start ← mid
   end if
end while
{* Obtaining the next hop *}
m1 ← NOT(prefixXORArray_Leaf[end].prefix)
if m1 < Array_Leaf[end].prefix_mask then
   next_hop ← Default next hop
else
   Match m1 against Path Information field of Array_Leaf[end] to get largest set bit
   next_hop ← Next hop corresponding to largest set bit
end if
```

information field. The longest matching prefix and the corresponding next hop address from the next hop list is picked. If no match is found, the default next hop address is returned.

6.2.1.3 Updating the Data Structure

Inserting or Deleting entries from the data space is equivalent to adding or deleting an entry from an array. To add an entry, a binary search is performed as outlined in the previous section, to find the location of the insertion. The entry is then added into the array, which is an O(N) process. The entry is also checked against entries above it to see if it is a subset or not, and the corresponding path information field and next hop list is updated. Deleting an entry follows a similar procedure. Updating the data structure, therefore, does not require the entire data structure to be built from scratch.

Table 6.5: Search space of the prefixes in Table 6.3

Prefix	Next Hop	Path Information	Next Hop List
00101*	7	0...010100	4
010110*	5	0...100000	-
011010*	6	0...100100	8
011100*	1	0...100100	8
101101*	2	0...101010	9,3
10111*	8	0...011010	9,3

6.2.2 Using Disjoint Prefixes for Binary Search

The search space used in the previous scheme can be reduced further by using only mutually disjoint prefixes. Two prefixes are considered disjoint, if neither of them is a prefix of the other. It is easy to see that these correspond to the leaf nodes of the tree shown in Figure 6.26. All internal nodes can be removed from the search space since the information corresponding to them is already contained in the path information field and the next hop list of the leaf nodes. The search space can then be shortened as shown in Table 6.5.

From Table 6.4 and Table 6.5 it might appear that a considerable amount of memory might be wasted in storing internal nodes a multiple number of times. For instance, next hop addresses corresponding to nodes 1011* and 10* are stored in the list of next hops for both 101101* and 10111*. This is not necessarily true. An examination of practical routing tables from [2] shows that most nodes in fact do not have any internal nodes and next hop lists to store. Figure 6.28 shows the number of internal nodes for all the leaf nodes for various routing tables. We note that more than 93% of the leaf nodes do not have any internal nodes in the path to the root node. Therefore, the overhead in memory to store internal nodes multiple times is actually quite small.

The build and search algorithms need to be modified slightly to accommodate the

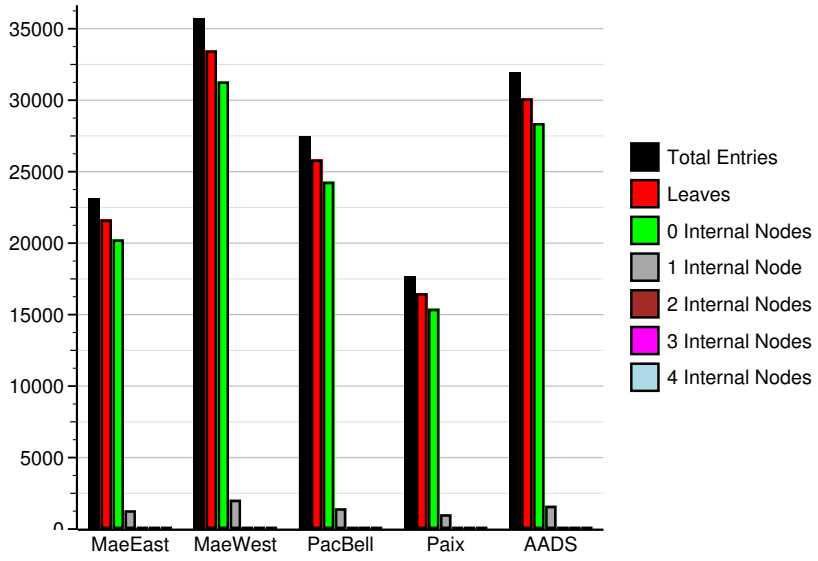


Figure 6.28: Profile of routing tables from [2]

changes and the modified algorithms are described in the following subsections.

6.2.2.1 Building the Data Structure

The pseudocode for building the data structure is given in Algorithm 5.

The main difference in building the data structure is that not all entries get added to the search space. If an entry is found to be a subset of the previous entry, then it is not added to the search space. The modified algorithm is described below:

Step 1 Each entry is read from the routing table and stored in an array.

Step 2 The entries are then sorted in an ascending order.

Step 3 Entries from the sorted list are then processed one at a time. Each entry is tested with the last of the leaf node entry added to see if it is a subset of the leaf node.

Step 3a If it is a subset, then the corresponding next hop information is added to the next hop list of the leaf node. The path information field of the leaf node is updated and so is the field containing the number of nodes. This step is then repeated for previous leaf node entries till the test for subset fails.

Algorithm 5 Building Procedure (Only Leaves)

{* *Array* refers to the initial array, $Array\_Leaf$ to the final *}
for $i = 1$ to N **do** {* N is the number of routing table entries *}
  $Array[i] \leftarrow$ Prefix Data
end for
Sort *Array*
$leaf\_ctr \leftarrow 0$
for $i = 1$ to N **do**
  if $Array[i].prefix$ is subset of $Array\_Leaf[leaf\_ctr].prefix$ **then** {* Insert in node list of $Array\_Leaf$ entry *}
    k $\leftarrow 0$
    while $Array[i].prefix$ is subset of $Array\_Leaf[leaf\_ctr-k].prefix$ **do** {* Check for all previous $Array\_Leaf$ entries too *}
      Update Path Information field of $Array\_Leaf[leaf\_ctr-k]$
      Increment the number of nodes field of $Array\_Leaf[leaf\_ctr-k]$
      Update $next\_hop$ list of $Array\_Leaf[leaf\_ctr-k]$
      Increment k
    end while
  else {* Add a new leaf to the $Array\_Leaf$ *}
    Increment $leaf\_ctr$
    Generate $Array\_Leaf[leaf\_ctr]$ field entries from $Array[i]$ field entries
  end if
end for

Step 3b If it is not a subset, then a new leaf node is added.

6.2.2.2 Searching the Data Structure

A binary search is performed on the leaf nodes, but the address now needs to be checked against both the upper and lower entries returned by the search. Algorithm 6 lists the pseudocode for searching a given prefix for its next hop address.

Algorithm 6 Search Procedure (Only Leaves)

```
{* Binary Search Part *}
start ← 1, end ← No_Leaves
while end > start + 1 do
   mid ← (start + end)/2
   if prefix < Array_Leaf[mid].prefix then
      end ← mid
   else
      start ← mid
   end if
end while
{* Obtaining the next hop *}
m1 ← NOT(prefix XOR Array_Leaf[end].prefix)
m2 ← NOT(prefix XOR Array_Leaf[end − 1].prefix)
if m1 > m2 then {* m1 is a better match *}
   if m1 < Array_Leaf[end].prefix_mask then
      next_hop ← Default next hop
   else
      Match m1 against Path Information field of Array_Leaf[end] to get largest set bit
      next_hop ← Next hop corresponding to largest set bit
   end if
else {* m2 is a better match *}
   if m1 < Array_Leaf[end − 1].prefix_mask then
      next_hop ← Default next hop
   else
      Match m1 against Path Information field of Array_Leaf[end − 1] to get largest set
      bit
      next_hop ← Next hop corresponding to largest set bit
   end if
end if
```

The steps involved in the search are listed below:

Step 1 Binary search of the address is performed on the leaf nodes which leads to a value

between two consecutive leaf node entries.

Step 2 The address is then matched with the two leaf nodes to see which of the leaf nodes is a better match.

Step 3 The better match from Step 2 is picked and checked against the path information field of the corresponding leaf node. The longest matching prefix and the corresponding next hop address from the next hop list is picked. If no match is found, the default next hop address is returned.

6.2.2.3 Updating the Data Structure

Inserting or deleting entries from the data structure does not require the search space to be built from scratch, as before. To add an entry, a binary search is performed to find the location of the insertion. If the entry turns out to be an internal node, only the next hop lists and the parent information fields of the corresponding leaf nodes get updated. If the entry to be added turns out to be a leaf node, then the leaf node array is updated which is an O(N) process. Deleting an entry follows a similar procedure.

CHAPTER 7

Performance and Analyses of Route Lookup Schemes

The route lookup schemes described in Chapter 6 were tested against practical routing tables from [2]. This chapter describes the performance of the schemes in comparison with other comparable schemes and also discusses the various performance related issues.

7.1 Performance of the Trie-Based Scheme

The address lookup part of the scheme was implemented in Verilog, since in an actual router this would be implemented in hardware. The generation of the SRAM and the DRAM data, which in a practical router would be performed in software, was written in C. A lookup can be performed every 64ns using this scheme. The amount of memory consumed for different routing tables is shown in Table 7.1. For instance, the MaeEast routing table with over 23,000 entries takes around 25KB of SRAM to store the bit pattern and around 12MB of DRAM to store the next hop addresses. In a conventional trie implementation, around 25MB of DRAM memory (the second last entry in the table) would be required. The last entry in the table shows the amount of compaction that can be achieved in the on-chip SRAM. For all the routing tables around 1 byte of SRAM memory per entry in the routing table is required. This gives very good scalability which would be

Table 7.1: Memory Requirements for various Routing Tables

Site	No of Entries	SRAM (KB)	DRAM (MB)	Trie Memory (MB)	Bytes/entry
MaeEast	23,113	24.4	11.43	24.28	1.08
MaeWest	35,752	34.75	16.32	34.683	1.99
PacBell	27,491	29.08	13.66	29.03	1.08
Paix	17,641	20.5	9.63	20.46	1.19
AADS	31,958	32.25	15.15	32.18	1.03

very important when routing tables become even larger in the future.

The overall compaction achieved in this scheme is much higher than other existing comparable schemes. The required SRAM is small enough (about 35KB for a routing database $>$30,000 entries) to easily fit on a chip. This is useful especially when moving to IPv6 where larger routing tables or multiple tables for different hierarchies would be used. The data in our case is compacted to around 1 byte for every entry in the routing table (for a 16-way trie). In comparison, the forwarding table by Degermark et al. [90] uses 5-6 bytes per entry. The implementation by Huang et al. [88] has an even larger forwarding table. Also, the overall memory consumption (SRAM and DRAM) using this scheme is almost half that required in conventional implementations. The static instruction count for building the SRAM and the DRAM data from the trie is 170 and the total CPU time taken to build this is in the order of 100ms on a Sun Ultra 5 with a 333 MHz processor. Since most forwarding tables need to be updated only about once every second, building the entire database from scratch is not an issue.

Our implementation requires 8 SRAM and 1 DRAM accesses per lookup. Splitting the SRAM and performing a direct lookup on the first 16 bits further reduces the number of accesses to 5 SRAM and 1 DRAM. This is easily pipelined so that the DRAM cycle time is the limiting factor. By implementing queues and multiple DRAMs in parallel, an even higher throughput can be obtained. In the current implementation with a single DRAM, a lookup can be done every 64ns which gives over 15 million lookups per second.

In a conventional implementation, 8 DRAM accesses would be required. DRAM accesses being quite expensive (60-65ns per random read/write as opposed to $<$10ns for SRAM) [57] the conventional implementation would be much slower than this scheme.

The amount of memory used in our scheme is more than the 3-4 MB typical of Patricia and basic binary schemes as in [1]. This is only because our scheme uses a 16-way trie in order to reduce the depth of the trie, and trie completion takes up extra memory. The advantages of using a 16-way trie is the reduction in depth, which leads to a smaller latency. There is more redundancy in the DRAM data as seen in Figure 6.14 but using extra off-chip DRAM memory in order to reduce DRAM accesses is a better alternative.

7.1.1 Analysis of the Scheme

Memory consumption depends on various factors. This section discusses the sensitivity of the total memory consumed to various parameters and also gives a theoretical analysis of the memory consumed.

7.1.1.1 Sensitivity of Performance to Degree of Trie

The overall throughput of the forwarding engine can be kept constant by altering the hardware FSM. The amount of SRAM consumed decreases, mainly due to less wastage in the trie completion step. The amount of SRAM required for various degrees of the trie is shown in Table 7.2. Efficiency of memory consumption increases with decreasing degree of the trie, i.e. fewer bytes per entry are required to store the SRAM data. This is due to less memory wastage in trie completion. The total latency of the address lookup changes with the degree of the trie. This is due to the fact that for smaller degrees, more SRAM accesses have to be made to traverse the trie since the depth of the trie increases. For instance, a degree 2 trie would require 16 stages to traverse down to the bottom of the trie structure. A more deeply pipelined FSM would be required to maintain the same throughput. We chose a 16-way trie in order to reduce the latency and to keep the

Table 7.2: SRAM requirements for different degrees of the trie structure

Site	No of Entries	Degree=16 KB(B/entry)	Degree=8 KB(B/entry)	Degree=4 KB(B/entry)	Degree=2 KB(B/entry)
MaeEast	23,113	24.4(1.08)	16(0.71)	8.09(0.36)	6.57(0.29)
MaeWest	35,752	34.75(1.99)	23.03(0.66)	11.41(0.33)	9.23(0.26)
PacBell	27,491	29.08(1.08)	19.24(0.72)	9.76(0.36)	7.99(0.3)
Paix	17,641	20.5(1.19)	13.09(0.76)	6.86(0.4)	5.6(0.33)
AADS	31,958	32.25(1.03)	21.33(0.68)	10.67(0.34)	8.67(0.28)

Table 7.3: SRAM and DRAM memory consumption for different degrees of the trie structure

Site	Degree=16		Degree=8		Degree=4		Degree=2	
	SRAM (KB)	DRAM (MB)	SRAM (KB)	DRAM (MB)	SRAM (KB)	DRAM (MB)	SRAM (KB)	DRAM (MB)
MaeEast	24.4	11.43	16	3.87	8.09	1	6.57	0.41
MaeWest	34.75	16.32	23.03	5.58	11.41	1.4	9.23	0.57
PacBell	29.08	13.66	19.24	4.66	9.76	1.2	7.99	0.5
Paix	20.5	9.63	13.09	3.17	6.86	0.84	5.6	0.35
AADS	32.25	15.15	21.33	5.17	10.67	1.31	8.67	0.54

hardware simple. This comes at the cost of higher memory consumption.

The amount of compaction from DRAM to SRAM decreases with decreasing degree of the tree as shown in Table 7.3. This is due to the fact that for a 16-way trie, information corresponding to a node with all 16 children is represented by a single bit in the SRAM. For a 4-way trie, information corresponding to only 4 children is represented by 1 bit. The SRAM data is about 500 times smaller than the DRAM data for a 16-way trie, 250 times for a 8-way trie, 125 times for a 4-way trie and about 64 times smaller for a 2-way trie.

7.1.1.2 Upper and Lower Bounds on the SRAM Required

The compaction achieved in the SRAM in our implementation is around 1Byte/entry as shown in Table 7.1. Compaction depends on the nature of the routing table and on the

Table 7.4: Routing Table for computing lower bound

Prefix	Next Hop
00*	1
01*	2
10*	3
11*	4

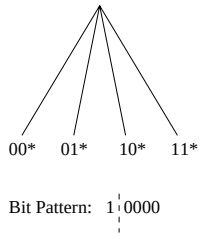


Figure 7.1: Trie structure for the entries in Table 7.4

degree of the trie used. In this section, we determine the bounds on the compaction that can be achieved in order to give an idea of the possible range of compaction. Computing the bounds on the SRAM memory is easy. Every time a node is added to the trie structure, it results in the addition of an M-bit pattern (where M is the degree of the trie) in the SRAM. This fact can be used to compute the upper and lower bounds on the memory. To compute the lower bound, consider the entries shown in Table 7.4. For a 4-way implementation, the trie structure and the bit pattern would be as shown in Figure 7.1. The first entry results in the addition of a node, whereas subsequent entries fit into the created node till the node is complete. Therefore, M entries could be represented by an M bit pattern or in other words, 1 bit per entry is the minimum amount of memory required (the 1 corresponding to the root node can be ignored, as before).

To compute the upper bound, we consider the case when an entry results in the addition of as many nodes as possible. The maximum number of nodes that can be added for any entry is equal to the depth of the trie. And since each node results in an M-bit pattern the upper bound on the memory required is $D.M$, where D is the depth of the trie and M is the degree of the trie. For example, consider the entry 0010 0010 ... 0010. This

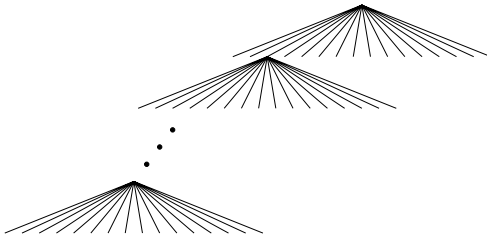


Figure 7.2: Trie structure for computing upper bound

32-bit entry results in the trie structure shown in Figure 7.2 and leads to 8 nodes being added to the 16-way trie. The amount of memory used in this example is 16 bytes/entry.

The SRAM memory therefore, lies between

$1\ bit/entry \leq SRAM\ Memory \leq D.M\ bits/entry$

For the 16-way implementation of the paper, the bounds on the memory is given by:

$1\ bit/entry \leq SRAM\ Memory \leq 16\ Bytes/entry$

The upper and lower bounds correspond to extreme cases and are not representative of practical routing tables. The lower bound assumes that the trie is complete whereas the upper bound case assumes that all entries are 32 bits wide (for IPv4) and don't share a common node along the path. This is hardly the case for practical routing tables which are usually sparse and share common nodes.

7.1.1.3 Expected SRAM Required

The upper and lower limits on the SRAM memory correspond to pathological cases which do not occur in real routing tables. This section determines the expected SRAM memory required, assuming completely random distribution of entries in the routing table.

Theorem 1 *The expected SRAM memory (bits/entry), for n random uniformly distributed entries, is given by*

$$E(Mem(Bits/entry)) = \frac{M}{ln(M)} \tag{7.1}$$

where M is the degree of the trie.

Proof: We start with the recurrence relation given in [75]. Let A_n be the average number of nodes in a random M-ary search trie that contains n keys. Then $A_0 = A_1 = 0$ and for $n \geq 2$,

$$A_n = 1 + \sum_{k_1+\ldots k_M=n} \left(\frac{n!}{k_1!\ldots k_M!} M^{-n}\right)(A_{k_1} + \ldots + A_{k_M}) \tag{7.2}$$

$\frac{n!M^{-n}}{k_1!\ldots k_M!}$ is the probability that k_1 of the keys are in the first sub-trie,…,k_M in the M^{th}. Using symmetry and summing over $k_2 \ldots k_M$, the above equation can be reduced to:

$$A_n = 1 + M^{1-n} \sum_{k=2}^{n} \binom{n}{k} (M-1)^{n-k} A_k \tag{7.3}$$

The remainder of the proof does not appear in [75]. However, it utilizes the techniques and methods developed in [75] to solve recurrence relations and determine their asymptotic values. To determine the expected memory, we first solve for the general recurrence relation given by:

$$A_n = C_n + M^{1-n} \sum_{k=2}^{n} \binom{n}{k} (M-1)^{n-k} A_k \tag{7.4}$$

To solve the above relation, we take the binomial transform of A_n, defined as:

$$\hat{A}_n = \sum_{k=2}^{n} \binom{n}{k} (-1)^k A_k \tag{7.5}$$

Substituting for A_k from (7.4), we get

$$\hat{A}_n = \sum_{k=2}^{n} \binom{n}{k} (-1)^k \left\{ C_n + M^{1-k} \sum_{l=2}^{k} \binom{k}{l} (M-1)^{k-l} A_l \right\} \tag{7.6}$$

or,

$$\hat{A}_n = \hat{C}_n + \sum_{k=2}^{n} \sum_{l=2}^{k} \binom{n}{k} (-1)^k M^{1-k} \binom{k}{l} (M-1)^{k-l} A_l \tag{7.7}$$

However,

$$\begin{aligned}\binom{n}{k}\binom{k}{l} &= \frac{n!}{k!(n-k)!}\frac{k!}{l!(k-l)!} \\ &= \frac{n!}{(n-l)!(n-k)!}\frac{(n-l)!}{l!(k-l)!} \\ &= \binom{n}{l}\binom{n-l}{k-l}\end{aligned} \tag{7.8}$$

Therefore, (7.7) can now be written as

$$\hat{A}_n = \hat{C}_n + \sum_{k=2}^{n}\sum_{l=2}^{k}\binom{n}{l}\binom{n-l}{k-l}(-1)^k M^{1-k}(M-1)^{k-l}A_l \tag{7.9}$$

Interchanging the order of summation, (7.9) reduces to

$$\hat{A}_n = \hat{C}_n + \sum_{l=2}^{n}\binom{n}{l}(-1)^l A_l \sum_{k=l}^{n}\binom{n-l}{k-l}(-1)^{k-l}M^{1-k}(M-1)^{k-l} \tag{7.10}$$

Introducing a new variable $j = k-l$, the above equation now reduces to

$$\begin{aligned}\hat{A}_n &= \hat{C}_n + \sum_{l=2}^{n}\binom{n}{l}(-1)^l A_l \sum_{j=0}^{n-l}\binom{n-l}{j}(-1)^j M^{1-j-l}(M-1)^j \\ &= \hat{C}_n + \sum_{l=2}^{n}\binom{n}{l}(-1)^l A_l M^{1-l}\sum_{j=0}^{n-l}\binom{n-l}{j}(-1)^j\left(\frac{M-1}{M}\right)^j\end{aligned} \tag{7.11}$$

The second summation constitutes the binomial expansion of $(1-(\frac{M-1}{M}))^{n-l}$ which is equal to $(1/m)^{n-l} = m^{l-n}$. Therefore, (7.11) reduces to the form

$$\begin{aligned}\hat{A}_n &= \hat{C}_n + \sum_{l=2}^{n}\binom{n}{l}(-1)^l M^{1-n}A_l \\ &= \hat{C}_n + M^{1-n}\sum_{l=2}^{n}\binom{n}{l}(-1)^l A_l\end{aligned} \tag{7.12}$$

Since $A_0 = A_1 = 0$, the summation on the RHS of (7.12) is also equal to $\hat{A}_n$. The above equation then simplifies to

$$\hat{A}_n = \left(\frac{M^{n-1}}{M^{n-1} - 1} \right) \hat{C}_n \tag{7.13}$$

Taking the binomial transform again,

$$\begin{aligned} A_n &= \sum_{k=2}^{n} \binom{n}{k} (-1)^k \frac{M^{k-1}}{M^{k-1} - 1} \hat{C}_k \\ &= \sum_{k=2}^{n} \binom{n}{k} (-1)^k \frac{M^{k-1} + 1 - 1}{M^{k-1} - 1} \hat{C}_k \\ &= C_n + \sum_{k=2}^{n} \binom{n}{k} (-1)^k \frac{1}{M^{k-1} - 1} \end{aligned} \tag{7.14}$$

The equation can now solved once $\hat{C}_k$ is determined. C_k corresponds to the sequence $< 0, 0, 1, \ldots, 1 >$ and the binomial transform of the sequence is easily shown to be $k-1$. (7.14) then finally reduces to

$$A_n = C_n + \sum_{k=2}^{n} \binom{n}{k} (-1)^k \frac{1}{M^{k-1} - 1} (k-1) \tag{7.15}$$

(7.15) gives the expected number of nodes in an M-ary trie storing n prefixes. Finally, we compute the asymptotic value of (7.15) by splitting (7.15) and computing the asymptotic value of each term as shown below.

$$\begin{aligned} A_n &= C_n + \sum_{k=2}^{n} \binom{n}{k} (-1)^k \frac{k}{M^{k-1} - 1} - \sum_{k=2}^{n} \binom{n}{k} (-1)^k \frac{1}{M^{k-1} - 1} \\ &= C_n + U_n + V_n \end{aligned} \tag{7.16}$$

V_n can be simplified as

$$V_n = k = 2^n \binom{n}{k} (-1)^k \sum_{j\geq 1} \left(\frac{1}{M^{k-1}}\right)^j = \sum_{j\geq 1} (M^j (1-M^{-j})^n - M^j + n) \tag{7.17}$$

By further setting $x = n/M^j$, V_n can be reduced as shown in [107] to:

$$V_n = \sum_{j\geq 1} \frac{n}{x}(e^{-x} - 1 + x) = n \sum_{j\geq 1} \frac{1}{x}(e^{-x} - 1 + x) \tag{7.18}$$

U_n can be simplified similarly. The extra factor k changes the factorial term to give:

$$U_{n+1} = \sum_{j\geq 1} (-1)(n+1)(e^{-x} - 1) = (n+1) \sum_{j\geq 1} (1 - e^{-x}) \tag{7.19}$$

Next we use a couple of standard results from complex variable theory as described in [107].

$$e^{-x} = \frac{1}{2\pi i} \int_{1/2-i\infty}^{1/2+i\infty} \Gamma(z) x^{-z} dz = \frac{1}{2\pi} \int_{-\infty}^{\infty} \Gamma(1/2 + it) x^{-(1/2+it)} dt \tag{7.20}$$

This can be shown to be equal to the sum of residues, which is

$$\sum_{0\leq k<M} x^{-k} \frac{(-1)^k}{k!} \tag{7.21}$$

Changing the limit of the integral in (7.20) is equivalent to removing the corresponding poles. The critical quantities in (7.18) and (7.19) can then be expressed as

$$\begin{aligned} \frac{1}{2\pi i} \int_{-3/2-i\infty}^{-3/2+i\infty} \Gamma(z) x^{-z} dz &= e^{-x} - 1 + x \\ \frac{-1}{2\pi i} \int_{-1/2-i\infty}^{-1/2+i\infty} \Gamma(z) x^{-z} dz &= 1 - e^{-x} \end{aligned} \tag{7.22}$$

(7.16) can then be written as (after placing the sum inside and simplifying)

$$A_n = C_n + \frac{n}{2\pi i}\int_{-3/2-i\infty}^{-3/2+i\infty}\frac{\Gamma(z)n^{-1-z}}{M^{-1-z}}dz - \frac{(n+1)}{2\pi i}\int_{-1/2-i\infty}^{-1/2+i\infty}\frac{\Gamma(z)n^{-z}}{M^{-z}}dz \tag{7.23}$$

Calculating the residues at the poles as shown in [107], the asymptotic value can be shown to be

$$\begin{aligned} A_n &= O(1) + \left(nlog_m n + \frac{n\gamma}{ln(m)} - n/2 - nf_0(n-1) + O(1)\right) \\ &\quad - \left(nlog_m n + \frac{n\gamma}{ln(m)} - \frac{n}{ln(m)} - n/2 + nf(n) + O(1)\right) \\ &= \frac{n}{ln(m)} + (negligible\ terms) + O(1) \end{aligned} \tag{7.24}$$

Since each node in the trie gives rise to M bits in the SRAM, the expected SRAM consumption (ignoring the smaller terms) is:

$$E(Mem(Bits/entry)) = \frac{M}{n}\frac{n}{ln(M)} = \frac{M}{ln(M)} \tag{7.25}$$

■

From the above analysis, the expected SRAM consumption can be calculated for the different routing tables. This is shown in Table 7.5. As before, the terms in parentheses represents SRAM memory expressed in Bytes/entry. The expected value gives a reasonably good prediction of the SRAM size requirement especially for lower degrees.

The main reason why the expected SRAM requirement is not very accurate is due to the fact that practical routing table entries are not really random. Most of the entries in the routing table have a prefix length of 16 or 24 which tends to skew the results. The $ln(M)$ term in (7.25) comes due to the fact that as the degree of the trie increases, more entries do not result in the addition of new nodes but get absorbed in already existing nodes. Due to the nature of the routing tables, this does not happen as often as it happens for random

Table 7.5: Expected SRAM requirements for different degrees of the trie structure

Site	No of Entries	Degree=16 KB(B/entry)	Degree=8 KB(B/entry)	Degree=4 KB(B/entry)	Degree=2 KB(B/entry)
MaeEast	23,113	16.25(0.72)	10.83(0.48)	8.13(0.36)	8.13(0.36)
MaeWest	35,752	25.14(0.72)	16.76(0.48)	12.57(0.36)	12.57(0.36)
PacBell	27,491	19.33(0.72)	12.89(0.48)	9.66(0.36)	9.66(0.36)
Paix	17,641	12.40(0.72)	8.27(0.48)	6.20(0.36)	6.20(0.36)
AADS	31,958	22.47(0.72)	15(0.48)	11.23(0.36)	11.23(0.36)

entries.

7.1.1.4 Sensitivity of Performance to SRAM width

A wider SRAM like 512 or 1024 bit-wide could be used in the design. This would not change the performance of the system but would reduce the memory overhead of the forwarding engine. In the current implementation, 20 bits are used to hold the sum of 1's value for every 128 bits of data in the SRAM row. The memory overhead is therefore 15-16%. By going to a design using 512 bit-wide SRAM, the memory overhead can be reduced to under 4%. The number of memory accesses would still remain the same. Using a wider SRAM in the design would require additional hardware to compute the sum of 1's, though the timing constraints would still be met in the current technology.

7.2 Performance of the Binary Based Scheme

To evaluate the performance of the binary schemes, we ran the algorithms on a Sun Ultra 5 with a 333 MHz processor and 512MB of RAM. The programs were written in C and compiled with gcc with the compiler optimization level set to 3. The results were also compared against the binary search implementation (LSV scheme) of Lampson et al. [1]. The binary search part of all algorithms was identical. The search time, build time and memory consumption were used to evaluate the performance of the schemes. Practical

Table 7.6: Average Search Times for Different Routing Tables

Routing Table	No of Entries	Binary Search (All nodes)	Binary Search (Only Leaves)	LSV Scheme
MaeEast	23113	662ns	610ns	761ns
MaeWest	35752	742ns	652ns	845ns
PacBell	27491	703ns	656ns	761ns
Paix	17641	640ns	634ns	739ns
AADS	31958	700ns	640ns	777ns

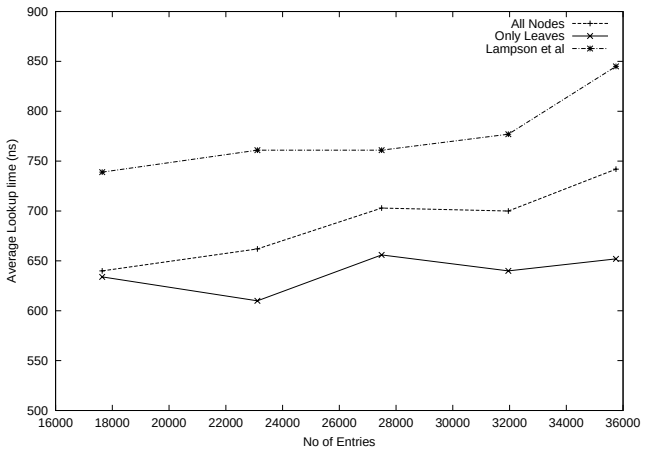


Figure 7.3: Average Lookup Time for Different Schemes

routing tables from [2] were used in the experiments.

1. *Average Search Time*: Random IP addresses were generated and a lookup was performed using both schemes. Table 7.6 lists the average lookup times for different routing tables. From Table 7.6 it can be seen that our scheme which uses all nodes in the search space results in over 10% improvement in lookup speeds as opposed to the LSV scheme. With internal nodes eliminated from the search space, an improvement of 15-20% can be obtained. The difference starts getting larger as the size of the routing table increases as seen from Figure 7.3.

2. *Build Time of Data Structure*: The time required to build the searchable structure from the routing tables is shown in Table 7.7. The time shown does not include the initial time taken to read entries from a file and store them in an array. This means

Table 7.7: Time Taken to Build Searchable Structure

Routing Table	No of Entries	Binary Search (All Nodes)	Binary Search (Only Leaves)	LSV Scheme
MaeEast	23113	80ms	80ms	210ms
MaeWest	35752	130ms	120ms	330ms
PacBell	27491	90ms	90ms	260ms
Paix	17641	60ms	50ms	150ms
AADS	31958	120ms	100ms	300ms

that for all algorithms time starts from sorting the entries and ends when the searchable structure is built. As seen from the table the time taken to build the searchable space in the LSV scheme is more than twice the time taken in our scheme. A profile of the times taken, using gprof, shows that most of the difference can be accounted for by the initial sorting of entries. Since the number of entries in the LSV scheme is more than twice the number of entries in ours, sorting becomes a fairly expensive operation. Build time for the scheme that uses only leaves in its searchable space is not significantly different from the one that uses all the nodes. This is due to the fact that most of the entries in these routing tables end up as leaves and the overhead in adding internal nodes is very small.

3. *Memory Consumption* Table 7.8 shows the memory required in storing the searchable structure for all schemes. Memory required for both binary schemes is close to half of that required in the LSV scheme. This is because each prefix in the LSV algorithm gives rise to two entries in the binary search table. Memory requirement for the scheme using only leaves in its searchable space is not significantly different from the one using all the nodes, for the reason pointed before.

Table 7.8: Memory Requirement for Different Routing Tables

Routing Table	No of Entries	Binary Search (All Nodes)	Binary Search (Only Leaves)	LSV Scheme
MaeEast	23113	0.62MB	0.58MB	1.06MB
MaeWest	35752	0.96 MB	0.9MB	1.64MB
PacBell	27491	0.74MB	0.7MB	1.26MB
Paix	17641	0.48MB	0.45MB	0.81MB
AADS	31958	0.86MB	0.81MB	1.46MB

7.2.1 Discussion and Analyses of the Binary Schemes

The performance of the binary schemes has been compared with the scheme by Lampson et al. [1] (referred to as the LSV scheme). The LSV scheme has been shown to perform better than many other schemes and it is also similar to our scheme in that it does a binary search on the entries in the routing table. In the LSV scheme, each entry in the routing table is expanded to two entries in the binary search space. The time taken to search for an entry is of the order of log(2N) where N is the number of entries in the routing table. Once the binary search is performed, an additional memory lookup is required (approximately half the time) to obtain the next hop address. In comparison, the search space in our schemes is $\leq N$. Our schemes therefore, result in 1-2 fewer memory accesses during the binary search. After the search is narrowed down, our schemes do not require any additional memory accesses to determine the next hop address in most cases. As a result, the average lookup time in these schemes is lower than the lookup time in LSV scheme.

The other main point of difference is the build time of the searchable structure. As pointed out earlier, the build time of these schemes is also much smaller than the build time of LSV scheme (see Table 7.7). The main reason for this difference is the sorting step. Sorting becomes more and more expensive as the number of entries to be sorted increases. The build time will be particularly important for larger routing tables, because updating entries requires the entire searchable space to be built from scratch in the LSV

scheme. For large routing tables (>100,000 entries), this could be a serious problem in the LSV scheme.

One potential problem with the binary schemes described is that the path information field has to be equal to the size of the address. For IPv4 this means that the path information field has to be 32 bits. For IPv6, this would mean storing a 128 bit field, leading to higher memory consumption and longer times to process instructions using this 128 bit field. In comparison, the additional information that the LSV scheme uses are pointers, the size of which would depend on the number of entries. For a routing table of 100,000 entries, the pointers need to be only 18 bits wide. Storing the high and low pointers in their scheme would bring the extra memory consumption to 36 bits as compared to 128 bits in our case. However, if very long or very short prefixes do not exist in the routing tables, then the number of bits used in storing the path information can be reduced. Figure 7.4 shows the various possible prefix lengths and the corresponding number of entries for the MaeEast routing table from [1]. The figure shows that no prefixes exist for prefix lengths smaller than 8 bits or larger than 30 bits. The path information field, therefore, only needs to have 23 bits to store the relevant information. At present no data exists which shows what core routing tables would look like for IPv6, but one can expect to be able to considerably reduce the number of bits used in storing the path information.

7.2.1.1 Average Memory Accesses

To compare the average memory accesses required by the different binary schemes, we break the total number of memory accesses into (a) memory accesses for the binary search alone and (b) additional memory accesses after the search is narrowed down. The average number of comparisons for an unsuccessful search is given by [75]

$$(\lfloor log_2 N \rfloor + 2) - \frac{1}{N+1}(2^{\lfloor log_2 N \rfloor + 1}) \tag{7.26}$$

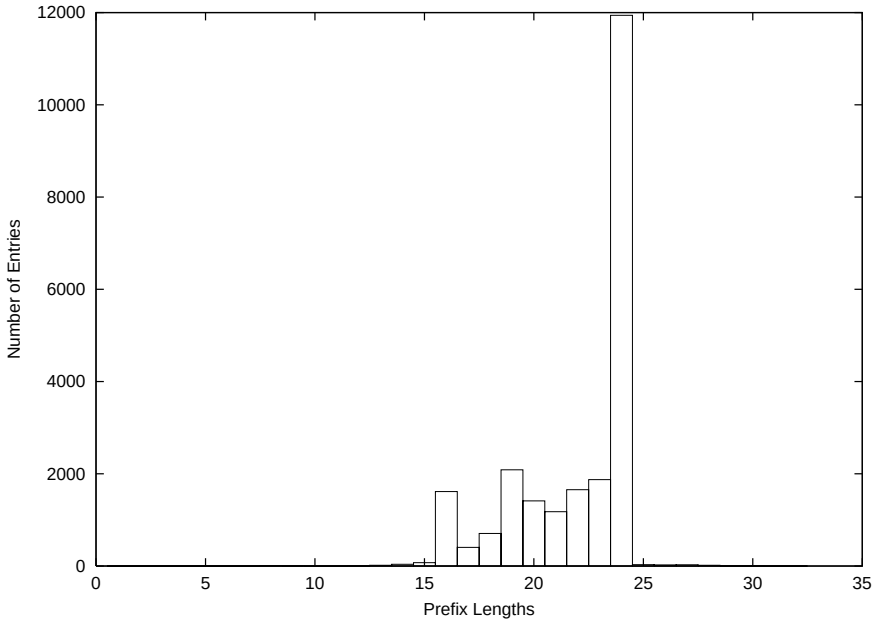


Figure 7.4: Prefix Length Distribution for the MaeEast Router

where N is the total number of entries. The case for an unsuccessful search is taken because most core routing tables do not actually contain complete destination addresses so a successful search never happens.

The average number of memory accesses for the different binary algorithms is given by:

$$\begin{aligned} A_{AllNodes} &= (\lfloor log_2 N \rfloor + 2) - \frac{1}{N+1}(2^{\lfloor log_2 N \rfloor + 1}) + K_1 \\ A_{OnlyLeaves} &= (N'+1)(\lfloor log_2 N' \rfloor + 2) - \frac{1}{N'}(2^{\lfloor log_2 N' \rfloor + 1}) + K_1 \\ A_{Lampson} &= (\lfloor log_2 (2N) \rfloor + 2) - \frac{1}{(2N)+1}(2^{\lfloor log_2 (2N) \rfloor + 1}) + K_2 \end{aligned} \tag{7.27}$$

where K_1 and K_2 are the average number of memory accesses required after the basic binary search narrows the search to between two consecutive entries. They are therefore constants determined by the algorithm. For Lampson's scheme, K_2 is 0.5 because an additional memory access is required half the time to determine the next hop address. In contrast, K_1 is ≈ 0.07 since only about 7% of the time, an additional memory access is required. This is due to the fact that most of the entries do not carry any parent node

Table 7.9: Average Memory References Required for Different Routing Tables

Routing Table	Binary Search (All Nodes)	Binary Search (Only Leaves)	LSV scheme
MaeEast	14.652	14.556	16.083
MaeWest	15.237	15.112	16.667
PacBell	14.878	14.802	16.308
Paix	14.213	14.083	15.643
AADS	15.045	14.982	16.475

information. Also, $N' \leq N$, in (7.27). As can be seen from the equations, LSV scheme takes about 1.5 memory accesses more than the other two schemes. The average number of memory accesses for different routing tables is shown in Table 7.9.

CHAPTER 8

Conclusions and Future Work

This dissertation discussed the design of two memory intensive systems: an FFT engine for use in high performance DSP systems and forwarding engines used in high speed routers. Both systems are memory starved and some amount of re-engineering (from a memory perspective) resulted in improved performances. The main focus in re-engineering was to limit random DRAM accesses, which have a high time penalty. In the case of the FFT design, this was achieved by storing data in DRAM banks in a way that consecutive cycles do not access different rows in the same bank. In the case of routers, since the data access pattern is random, the aim was to limit the number of DRAM accesses that need to be made.

The FFT system design used efficient memory mapping schemes to avoid any precharge and refresh times associated with DRAMs. By using a novel scheme to generate twiddle factors on chip, memory requirement associated with storing the twiddle factors is reduced drastically. This was especially important for the large FFT size under consideration. Using these techniques, the I/O bandwidth provided by the SHOCC technology could be fully utilized. A complete million-point FFT could be performed in 1.31ms.

The FFT design discussed in this dissertation was of an ASIC nature. While this was sufficient to demonstrate ways to manage data efficiently, a more useful implementation would be of a programmable DSP engine. The same techniques could perhaps be used by the compiler in that case to make decisions regarding memory storage, scheduling operations etc. The decisions would depend on the application (e.g. FFT, QR factorization etc.) running at the time.

Our forwarding engine designs tried to limit the number of DRAM accesses. Two types of schemes were discussed: a binary-trie based scheme and binary searches. The total number of memory accesses in a trie based scheme depends on the address size. Therefore, this scheme would perform well for smaller address sizes as in IPv4. However, for IPv6 with 128-bit address size, the total lookup time would be too large. For large address sizes, binary search based schemes perform better since the lookup depends only on the number of entries in the routing table and not the size of the address.

In the trie-based scheme, a trie-compaction algorithm was used which could compress the trie-path information such that it is small enough to fit on an on-chip SRAM. The trie traversal can then be performed in SRAM, which is much faster than DRAMs. The amount of compaction achieved is also much larger than other existing schemes. For a 16-way trie, a routing table with over 23,000 entries takes only about 25KB of SRAM memory. A compaction of about 1 byte per entry was achieved in the SRAM. After the trie traversal, only a single DRAM access is required to determine the next hop address which could give a throughput of over 15 million lookups per second. In binary search schemes, the number of memory accesses was reduced by using an additional field (which contained the path information) along with the entries. The number of memory accesses required was O(log(N)) as opposed to O(log(2N)) of the binary scheme in the literature. This resulted in a speed improvement of 10-15%. A further improvement by using only disjoint prefixes in the data set enhanced the speed by up to 20%.

Various forwarding schemes have been suggested in recent literature, targeted at

various applications. A related problem in the design of high speed routers that this dissertation does not consider is that of packet classification [108]. This is required for implementing functions like QoS and firewalls. It changes the simple forwarding from a 1-dimensional problem to a K-dimensional problem, where K is the number of filter fields. A packet P matches a filter F if for all packet fields i, $P[i]$ matches $F[i]$. The problem of packet classification is to find the lowest cost filter matching for a given packet P. That is a challenging problem especially at high speeds and the schemes presented in this dissertation would probably not perform well in this scenario.

BIBLIOGRAPHY

[1] B. Lampson, V. Srinivasan, and G. Varghese. IP Lookups using Multiway and Multicolumn Search. In *Proc. IEEE INFOCOM'98*, volume 3, pages 1248–1256, San Francisco, CA, 1998.

[2] Michigan University and Merit Network. Internet Performance Management and Analysis (IPMA) Project. (http://nic.merit.edu/ ipma).

[3] Patrick P. Gelsinger. Microprocessors for the New Millenium: Challenges, Opportunities, and New Frontiers. In *IEEE International Solid-State Circuits Conference*, pages 22–25, 2001.

[4] M. Dibbs, P. Garrou, C.C. Chau, Y. So, D. Frye, J. Wagner, J. Ousley, G. Baugher, J. Santandrea, G. Connor, J. MacPherson, G. Adema, P. Dean, L.W. Schaper, R. Eden, and R. Sands. Development of Seamless High Off-Chip Connectivity. In *SPIE, Proc. Int. Symp. Microelectron.*, volume 3235, pages 138–143, October 1997.

[5] L. Schaper, M. Dibbs, P. Garrou, C.C. Chau, Y. So, D. Frye, J. Wagner, J. Ousley, G. Baugher, R. Pickard, G. Connor, D. Winn, and P. Deane. Seamless High Off-Chip Connectivity. In *IEEE*, pages 39–44, 1998.

[6] Peyman Dehkordi, Tim Powell, and Donald Bouldin. Performance Comparison of MCM-D and SMT Packaging Technologies for a DSP Subsystem. In *IEEE International Symposium on Circuits and Systems*, pages 245–248, 1996.

[7] Robert K. Scannell and John K. Hagge. Development of a Multichip Module DSP. *Computer*, 26:13–21, 1993.

[8] Michael Gdula, Kenneth B. Welles II, and Robert J. Wojnarowski. An 80Mhz Digital Signal Processing Multichip Module made with General Electric High Density Interconnect Technology. In *Proceedings of 3rd International Conference on Wafer Scale Integration*, pages 192–198, 1991.

[9] Toshio Sudo. Silicon-on-Silicon technology for CMOS-based computer systems. In *Proceedings of 1992 IEEE Multi-Chip Module Conference*, pages 8–11, 1992.

[10] Michel Michaud and Jean Claude Rames. Development of a DSP Function Using MCM Technology. In *Proceedings of 1994 IEEE Multi-Chip Module Conference*, pages 378–383, 1994.

[11] Tsuyoshi Isshiki and Wayne Wei-Ming Dai. Field-Programmable Multi-Chip Module (FPMCM) for High-Performance DSP Accelerator. In *1994 IEEE Asia-Pacific Conference on Circuits and Systems*, pages 139–144, 1994.

[12] Peyman Dehkordi, Tim Powell, and Donald Bouldin. Development of a DSP/MCM Subsystem Assessing Low-Volume, Low-cost MCM Prototyping for Universities. In *Proceedings of 1996 IEEE Multi-Chip Module Conference*, pages 89–94, 1996.

[13] Richard G. Rozier and Fouad E. Kiamilev. Design of an MCM FFT Processor. In *1997 IEEE Multi-Chip Module Conference*, pages 83–88, 1997.

[14] Hideki Yoshizawa, Tohru Tsuruta, Norichika Kumamoto, and Masanori Kurita. A High Performance DSP Architecture ”MSPM” for Digital Image Processing Using Embedded DRAM ASIC Technologies. In *The first IEEE Asia Pacific Conference on ASICs*, pages 413–416, 1999.

[15] K. Balmer, N. Ing-Simmons, P. Moyse, I. Robertson, J. Keay, M. Hammes, E. Oakland, R. Simpson, G. Barr, and D. Roskell. A Single Chip Multimedia Video Processor. In *Proceedings of the IEEE 1994 Custom Integrated Circuits Conference*, pages 91–94, 1994.

[16] Glenn D. Bergland. Fast Fourier Transform Hardware Implementations-An Overview. *IEEE Transactions on Audio and Electroacoustics*, AU-17(2):104–108, June 1969.

[17] Peter Pirsch. *Architectures for Digital Signal Processing*, chapter 7. John Wiley & Sons, 1998.

[18] D. R. Bungard, L. Lau, and T. L. Rorabaugh. New Programmable FFT Implementation for Radar Signal Processing. In *IEEE International Symposium on Circuits and Systems ’89*, pages 1323–1327, 1989.

[19] D. R. Bungard, L. Lau, and T. L. Rorabaugh. Programmable FFT Processors for Wide-Bandwidth HF Spread-Spectrum Communications and Radar Signal Processing. In *International Conference on Acoustics, Speech and Signal Processing*, pages 1357–1359, 1989.

[20] Lawrence R. Rabiner and Bernard Gold. *Theory and Application of Digital Signal Processing*, chapter 10. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1975.

[21] Ben Gold and Theodore Bially. Parallelism in Fast Fourier Transform Hardware. *IEEE Transactions on Audio and Electroacoustics*, AU-21(1):5–16, February 1973.

[22] E. Bernard, J. G. Krammer, M. Sauer, and R. Schweizer. A Pipeline Architecture for Modified Higher Radix FFT. In *1992 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 5, pages 617–620, 1992.

[23] E. Bidet, C. Joanblanq, and P. Senn. A Fast Single Chip Implementation of 8192 COmplex Points FFT. In *IEEE 1994 Custom Integrated Circuits Conference*, pages 207–210, 1994.

[24] E. Bidet, D. Castelain, C. Joanblanq, and P. Senn. A Fast Single Chip Implementation of 8192 COmplex Points FFT. *IEEE Journal of Solid State Circuits*, 30:300–305, March 1995.

[25] T. Widhe, J. Melander, and L. Wanhammar. Design of Efficient Radix-8 Butterfly PEs for VLSI. In *1997 IEEE International Symposium on Circuits and Systems*, pages 2084–2087, June 1997.

[26] Shousheng He and Mats Torkelson. A New Approach to Pipeline FFT Processor. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 766–770, 1996.

[27] Mark A. Richards. On HArdware Implementation of the Split-Radix FFT. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 36(10):1575–1581, October 1998.

[28] Jeremy R. Johnson Pinit Kumhom and Prawat Nagvajara. Design, Optimization, and Implementation of a Universal FFT Processor. In *Proceedings of the 13th Annual IEEE International ASIC/SOC Conference*, pages 182–186, 2000.

[29] R. Radhouane, P. Liu, and C. Modlin. Minimizing the Memory Requirement for Continuous Flow FFT Implementation: Continuous Flow Mixed Mode FFT (CFMM-FFT). In *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems*, volume 1, pages 116–119, 2000.

[30] Jae Sung Lee, Young Seop Jeon, and M.H. Sunwoo. Design of new DSP instructions and their hardware architecture for high-speed DSP. In *2001 IEEE Workshop on Signal Processing Systems*, pages 80–90, 2001.

[31] F. Franchetti, H. Karner, S. Kral, and C.W. Ueberhuber. Architecture Independent short vector FFTs. In *Proceedings of the 2001 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 2, pages 1109–1112, 2001.

[32] G. Bi and E.V. Jones. A Pipelined FFT Processor for Word Sequential Data. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37:1982–1985, December 1989.

[33] Shousheng He and Mats Torkelson. Design and implementation of a 1024-point pipeline FFT processor. In *Proceedings of the IEEE 1998 Custom Integrated Circuits Conference*, pages 131–134, 1998.

[34] Shousheng He and Mats Torkelson. Designing pipeline FFT processor for OFDM (de)modulation. In *Proceedings of the 1998 International Symposium on Signals, Systems, and Electronics*, pages 257–262, 1998.

[35] Bum Sik Kim and Lee-Sup Kim. Low Power Pipelined FFT Architecture for Synthetic Aperture Radar Signal Processing. In *IEEE 39th Midwest symposium on Circuits and Systems*, volume 3, pages 1367–1370, 1996.

[36] C.C.W. Hui, T.J. Ding, J.V. McCanny, and R.F. Woods. A 64-point Fourier transform chip for video motion compensation using phase correlation. *IEEE Journal of Solid State Circuits*, 31:1751–1761, November 1996.

[37] Kevin J. McGee. Comments on "A 64-point Fourier transform chip for video motion compensation using phase correlation". *IEEE Journal of Solid State Circuits*, 33(6):928–932, June 1998.

[38] K. Yamashita, A. Kanasugi, S. Hijiya, G. Goto, N. Matsumura, and T. Shirato. A wafer-scale 170000-gate FFT processor with built-in test circuits. *IEEE Journal of Solid State Circuits*, 23:336–342, April 1988.

[39] Alvin M. Despain. Very Fast Fourier Transform Algorithms for Hardware Implementation. *IEEE Transactions on Computers*, 28:333–341, May 1979.

[40] Cheng-Shing Wu and An-Yeu Wu. Modified Vector Rotational CORDIC (MVR-CORDIC) Algorithm and its Application to FFT. In *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems*, volume 4, pages 529–532, 2000.

[41] Jin-Fu Li, Shyue-Kung Lu, Shih-Arn Hwang, and Cheng-Wen Wu. Easily Testable and Fault-Tolerant FFT Butterfly Networks. *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, 47:919–929, September 2000.

[42] Jin-Fu Li and Cheng-Wen Wu. Testable and Fault-Tolerant Design for FFT Networks. In *International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 201–209, 1999.

[43] Shyue-Kung Lu, Jen-Sheng Shih, and Cheng-Wen Wu. A testable/fault-tolerant FFT Processor Design. In *Proceedings of the Ninth Asian Test Symposium, 2000*, pages 429–433, 2000.

[44] C.-W. Wu S.-K. Lu and S.-Y. Kuo. Enhancing testability of VLSI arrays for fast Fourier transform. *IEE Proc. Part E*, 140:161–166, May 1993.

[45] Marshall C. Pease. Organization of Large Scale Fourier Processors. *Journal of the ACM*, 16:474–482, July 1969.

[46] Danny Cohen. Simplified Control of FFT Hardware. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-24:577–579, December 1976.

[47] L.G. Johnson. Conflict Free Memory Addressing for Dedicated FFT Hardware. *IEEE Transactions on Circuit and Systems-II: Analog and Digital Signal Processing*, 39(5):312–316, May 1992.

[48] Yutai Ma. An Effective Memory Addressing Scheme for FFT Processors. *IEEE Transactions on Signal Processing*, 47(3):907–911, March 1999.

[49] Yutai Ma and Lars Wanhammar. A Hardware Efficient Control of Memory Addressing for High-Performance FFT Processors. *IEEE Transactions on Signal Processing*, 48:917–921, March 2000.

[50] Hose Antonio Hidalgo, Juan López, Fransisco Argüello, and Emilio L. Zapata. Area-Efficient Architecture for Fast Fourier Transform. *IEEE Transactions on Circuit and Systems-II: Analog and Digital Signal Processing*, 46(2):187–193, February 1999.

[51] Hsin-Fu Lo, Ming-Der Shieh, and Chien-Ming Wu. Design of an efficient FFT Processor for DAB System. In *The 2001 IEEE International Symposium on Circuits and Systems*, volume 4, pages 654–657, 2001.

[52] David T. Harper. Block, Multistride Vector, and FFT Accesses in Parallel Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):43–51, January 1991.

[53] David T. Harper. Conflict-Free Vector Access Using a Dynamic Storage Scheme. *IEEE Transactions on Parallel and Distributed Systems*, 40(3):276–283, March 1991.

[54] R. Bernardini, G.M. Cortelazzo, and G.A. Mian. A New Techinique for Twiddle-Factor Elimination in Multidimensional FFT's. *IEEE Transactions on Signal Processing*, 42(8):2176–2178, August 1994.

[55] M. Hasan and T. Arslan. Coefficient Memory Addressing Scheme for High Performance FFT Processors. *Electronics Letters*, 37:1322–1324, October 2001.

[56] M. Nakkar, A.W. Glaser, P. Franzon, K. Williams, M. Roberson, and G. Rinne. Three Dimensional MCM Package Assembly and Analysis. In *Proceedings of the IEEE/IMAPS Conference on High Density Packaging and MCMs*, pages 188–192, 1999.

[57] 128Mb DDR SDRAM Datasheet. (http://www.micron.com/products/datasheets/ddrsdramds.html).

[58] Nebojša Milenković, Vladimir Stanković, and Aleksandar Dimitrijvić. High Bandwidth DRAM Memories for DSP. *4th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services*, 2:502–505, 1999.

[59] Peter Gillingham and Bill Vogley. SLDRAM: High-Performance Open-Standard Memory. *IEEE Micro*, pages 29–39, 1997.

[60] ARS Technica Ram Guide, Part III: DDR DRAM and Rambus. (http://arstechnica.com/paedia/r/ram_guide/ram_guide.part3-1.html).

[61] Rambus DRAM: Uncovering facts & Burying Rumors. (http://www.anandtech.com/showdoc.html?i=1239).

[62] K. Sapiecha and R. Jarocki. Modular Architecture for High Performance Implementation of the FFT Algorithm. *IEEE Transactions on Computers*, 39(12):1464–1468, December 1990.

[63] Lawrence R. Rabiner and Bernard Gold. *Theory and Application of Digital Signal Processing*, chapter 6. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1975.

[64] http://bopsnet.com/cores.

[65] R. Crandall and J. Klivington. Supercomputer-style FFT library for Apple G4. (http://www.motorola.com/), 1999.

[66] W.J. Dally and J.W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

[67] T. Arabi, J. Jones, G. Taylor, and D. Riendeau. Modeling, Simulation, and Design Methodology of the Interconnect and Packaging of an Ultra-High Speed Source Synchronous Bus. In *IEEE 7th Topical Meeting on Electrical Performance of Electronic Packaging*, pages 8–11, 1998.

[68] Maxwell Q-3D User Guide. Ansoft Corporation.

[69] S. Afonso, W. D. Brown, L. W. Schaper, and J. P. Parkerson. Signal Propagation on Seamless High Off-Chip Connectivity. In *IEEE 7th Topical Meeting on Electrical Performance of Electronic Packaging*, pages 31–34, 1998.

[70] S. Afonso, L. W. Schaper, J. P. Parkerson, W. D. Brown, S. Ang, and H. A. Naseem. Modeling and Electrical Analysis of Seamless High Off-Chip Connectivity (SHOCC) Interconnects. *IEEE Transactions on Advanced Packaging*, 22(3):309–320, August 1999.

[71] Ron K. Poon. *Computer Circuits Electrical Design*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1995.

[72] A. Deutsch, G. V. Kopcsay, C. W. Surovic, B. J. Rubin, L. M. Terman, R. P. Dunne Jr., T. A. Gallo, and R. H. Dennard. Modeling and characterization of long on-chip interconnections for high-performance microprocessors. *IBM Journal of Research and Development*, 39(5):547–566, September 1995.

[73] Jan M. Rabaey. *Digital Integrated Circuits: A Design Perspective*, chapter 8. Prentice-Hall Inc., 1996.

[74] Pronita Mehrotra, Vikram Rao, Tom Conte, and Paul Franzon. Leveraging High Density Packaging for High Performance DSP Systems. In *IEEE 9th Topical Meeting on Electrical Performance of Electronic Packaging*, 2000.

[75] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Searching and Sorting*. Addison-Wesley Publishing Company, 1973.

[76] W. Richard Stevens and Gary R. Wright. *TCP/IP Illustrated, vol.2 - The Implementation*. Addison Wesley Publishing, 1995.

[77] Masanori Uga and Kohei Shiomoto. A fast and compact longest match prefix look-up method using pointer cache for very long network address. In *Proc. of the 8th International Conference on Computer Communications and Networks*, pages 595–602, 1999.

[78] Keith Sklower. A Tree-Based Routing Table for Berkeley Unix. In *Technical Report*, University of California, Berkeley.

[79] W. Doeringer, Günter Karjoth, and Mehdi Nassehi. Routing on Longest-Matching Prefixes. *IEEE/ACM Transactions on Networking*, 4(1):86–97, February 1996.

[80] E. Filippi, V. Innocenti, and V. Vercellone. Address Lookup Solutions For Gigabit Switch/Router, 1998.

[81] S. Nilsson and G. Karlsson. IP-Address Lookup Using LC-Tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, June 1999.

[82] S. Nilsson and M. Tikkanen. Implementing a dynamic compressed trie. In *Proc. 2nd Workshop on Algorithm Engineering (WAE '98), Saarbruecken, Germany, August 20-22, 1998.*, 1998.

[83] T. Pei and C. Zukowski. Putting Routing Tables in Silicon. *IEEE Network Magazine*, pages 42–50, January 1992.

[84] http://www.netlogicmicro.com/pressreleases/pr_11-1-99.html.

[85] V. Srinivasan and George Varghese. Faster IP Lookups using Controlled Prefix Expansion. In *ACM SIGMETRICS*, pages 1–10, Madison, WI, 1998.

[86] P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *Proc. IEEE INFOCOM'98*, pages 1382–1391, San Francisco, CA, 1998.

[87] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *Proc. ACM SIGCOMM*, volume 27, pages 25–36, October 1997.

[88] Nen-Fu Huang and Shi-Ming Zhao. A Novel IP-Routing Lookup Scheme and Hardware Architecture for Multigigabit Switching Routers. *IEEE Journal on Selected Areas in Communications*, 17(6):1093–1104, June 1999.

[89] Henry Hong-Yi Tzeng and Tony Przygienda. On Fast Address-Lookup Algorithms. *IEEE Journal on Selected Areas in Communications*, pages 1067–82, June 1999.

[90] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding tables for fast routing lookups. In *Proc. ACM SIGCOMM*, volume 27, pages 3–14, October 1997.

[91] H. Tzeng. Longest Prefix Search using Compressed Trees. In *Proceedings of IEEE Globecom, Sydney, Australia, November 8–12 1998.*, 1998.

[92] Nasser Yazdani and Paul S. Min. Fast and Scalabe schemes for the IP address Lookup Problem. In *Proc. IEEE Conference on High Performance Switching and Routing*, pages 83–92, 2000.

[93] Girish P. Chandranmenon and George Varghese. Trading Packet Headers for Packet Processing. *IEEE/ACM Transactions on Networking*, 4(2):141–152, April 1996.

[94] P. Newman, T. Lyon, and G. Minshall. Flow labelled IP: a connectionless approach to ATM. In *Proc. IEEE INFOCOM'96*, pages 1251–1260, 1996.

[95] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. Internet Draft, Apr. 1999.

[96] Tzi cker Chiueh and Prashant Pradhan. High-Performance IP Routing Table Lookup Using CPU Caching. In *Proc. IEEE INFOCOM'99*, pages 1421–1428, 1999.

[97] Tzi cker Chiueh and Prashant Pradhan. Cache Memory Design for Network Processors. In *Proceedings of Sxth International Symposium on High-Performance Computer Architecture, 2000*, volume HPCA-6, pages 409–418, 2000.

[98] D. Knox and S. Panchanathan. Parallel Searching Techniques for Routing Table Lookup. In *Proc. IEEE INFOCOM'93*, pages 1400–1405, San Francisco, CA, 1993.

[99] Gene Chuang and Steve McCanne. Optimal Routing Table Design for IP Address Lookup Under Memory Constraints. In *Proc. IEEE INFOCOM'99*, volume 3, pages 1437–1444, 1999.

[100] Pronita Mehrotra and Paul Franzon. Novel Hardware Architecture for Fast Address Lookups. In *Accepted for presentation in High Performance Switching and Routing*, May 2002.

[101] Pronita Mehrotra, Ilia Baldine, Dan Stevenson, and Paul Franzon. Leveraging High Density Packaging for High Performance DSP Systems. In *14th International ASIC/SOC Conference*, 2001.

[102] Marco Listanti, Vincenzo Eramo, and Roberto Sabella. Architectural and Technological Issues for Future Optical Networks. *IEEE Communications Magazine*, 38(9):82–92, September 2000.

[103] John Y. Wei and Ray I. McFarland. Just-in-Time Signaling for WDM Optical Burst Switching Networks. *Journal of Lightwave Technology*, 18:2019–2037, December 2000.

[104] Nick McKeown. Scalability of IP routers. In *Optical Fiber Communication Conference*, March 2001.

[105] Lakovos Mavroidis. A Low Power 200 MHz Multiported Register File for the Vector-IRAM chip. (http://www.cs.berkeley.edu/ maurog/report.pdf).

[106] Byoung-Hoon Lim and Jin-Ku Kang. A Self-Timed Pipelined Adder Using Data Align Method. In *The Second IEEE Asia Pacific Conference on ASICs*, pages 77–80, August 2000.

[107] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Searching and Sorting*, pages 131–134. Addison-Wesley Publishing Company, 1973.

[108] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification using Tuple Space Search. In *SIGCOMM*, 1999.