Title:
Implementation and simulation of the two-level lookup

Author:
Loukissas, Alexander

Acceptance Date:
2008

Series:
UC San Diego Electronic Theses and Dissertations

Degree:
M.S., Computer scienceUC San Diego

Permalink:
http://escholarship.org/uc/item/7hm4d6vt

Local Identifier:
b6636173

Abstract:
Today's data centers consist of several thousand PCs that provide massive amounts of computational power and storage capacity in a cost-effective manner. Due to the highly distributed nature of the applications running in these clusters, intra-node communication bandwidth is key to the performance of the data center. Unfortunately, communication bandwidth is in many cases a significant bottleneck in large-scale clusters. Although current data center networks leverage high-end switching elements with massive switching capacity, current architectures are heavily oversubscribed and fail to meet with the requirements in intra-node communication bandwidth. What is worse, performance greatly degrades with larger cluster sizes, while the cost increases exponentially with cluster size. This thesis presents the implementation of a network architecture that leverages commodity Ethernet switches and supports full bisection bandwidth in clusters consisting of tens of thousands of hosts. Together with Mohammad Al-Fares and Amin Vahdat, we have shown that by appropriately interconnecting switches and using a novel routing algorithm, we can achieve significantly better performance than current high-end solutions at a fraction of the cost. This architecture is fully compatible with Ethernet and applications running on TCP/IP, and requires no end-host modifications. The focus of this thesis is the implementation of our architecture in hardware. More specifically, I show that our architecture can be implemented in commodity switches in a straightforward manner, requiring only slight modifications in their hardware. The thesis also focuses on the implementation and evaluation of the architecture in a network simulator

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Implementation and Simulation of the Two-Level Lookup

A thesis submitted in partial satisfaction of the requirements for the degree
Master of Science

in

Computer Science

by

Alexander Loukissas

Committee in charge:

Amin Vadhat, Chair
Stefan Savage
George Varghese

2008

The thesis of Alexander Loukissas is approved and it is acceptable in quality and form for publication on microfilm:

____

____

____
Chair

University of California, San Diego

2008

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Amin Vahdat for his support as my advisor and the chair of my committee. Without his guidance and vision, this work would not have been as successful.

I would also like to acknowledge Mohammad Al-Fares, who has been a wonderful colleague and has made invaluable contributions to this project.

Several chapters in this thesis are extended versions of a paper titled ”A Scalable, Commodity Data Center Network Architecture” that I co-authored with Mohammad Al-Fares and Professor Amin Vahdat, and which appears in the proceedings of the 2008 ACM SIGCOMM conference.

More specifically, Chapter 1 is an adaptation of Section 1 of the paper, part of which is due to Professor Vahdat. The biggest part of Chapters 2 and 3 are the products of my research and an abbreviated version of them appears in the paper. Chapter 2 also presents some key aspects of the system architecture as described in the paper, which is joint work between Mohammad Al-Fares, Amin Vahdat and myself.

Further, I would like to acknowledge Professors Stefan Savage and George Varghese for their guidance and for serving in my thesis committee.

Finally, I would like to acknowledge Mary ET Boyle and the Cognitive Science department at UCSD for providing me with the financial support during the biggest part of my graduate studies.

ABSTRACT OF THE THESIS

Implementation and Simulation of the Two-Level Lookup

by

Alexander Loukissas
Master of Science in Computer Science

University of California, San Diego, 2008

Amin Vadhat, Chair

Today's data centers consist of several thousand PCs that provide massive amounts of computational power and storage capacity in a cost-effective manner. Due to the highly distributed nature of the applications running in these clusters, intra-node communication bandwidth is key to the performance of the data center.

Unfortunately, communication bandwidth is in many cases a significant bottleneck in large-scale clusters. Although current data center networks leverage high-end switching elements with massive switching capacity, current architectures are heavily oversubscribed and fail to meet with the requirements in intra-node communication bandwidth. What is worse, performance greatly degrades with larger cluster sizes, while the cost increases exponentially with cluster size.

This thesis presents the implementation of a network architecture that leverages commodity Ethernet switches and supports full bisection bandwidth in clusters consisting of tens of thousands of hosts. Together with Mohammad Al-Fares and Amin Vahdat, we have shown that by appropriately interconnecting switches and using a novel routing algorithm, we can achieve significantly better performance than current high-end solutions at a fraction of the cost. This architecture is fully compatible with Ethernet and applications running on TCP/IP, and requires no end-host modifications.

The focus of this thesis is the implementation of our architecture in hardware. More specifically, I show that our architecture can be implemented in commodity switches in a straightforward manner, requiring only slight modifications in their hardware. The thesis also focuses on the implementation and evaluation of the architecture in a network simulator.

Chapter 1

Introduction

Many of today's data centers consist of a few thousand PCs and are able to provide the required processing power and storage in a much more cost-efficient manner than supercomputer clusters have been able to. However, a significant problem in current data center design is providing high communication bandwidth between hosts. Current solutions that build upon common communication standards (Ethernet and IP) suffer from poor performance, while incurring a significant cost. Although there exist alternative network architectures that are able to provide high levels of bandwidth and scale to thousands of hosts, they leverage non-standard architectures that are not natively compatible with communication protocols and hardware.

Communication bandwidth greatly affects the total performance of the data center, since most of the applications are distributed among a large number of hosts and require frequent large data transfers across the network. For example, these traffic patterns are common before the reduce phase of a MapReduce [15] job, in distributed file systems [21], as well as in parallel scientific computations.

Today, the prevailing trend in data center design is to leverage a large number of commodity parts (e.g. PCs, storage devices), which can deliver the required computational and storage capacities, while helping minimize the total cost. On the other hand, the communication networks that are used to interconnect these clusters comprise of largely non-commodity parts, simply because one such solution today can neither provide the necessary communication bandwidth nor allow for large clusters to be built. However, trying to solve a quite similar problem for the telephone network, in 1953 Charles Clos proposed a way to interconnect small, low-cost switches to build a practical multi-stage telephone switching system [13]. In fact, this topology has been the basis for many interconnects in massively parallel processing (MPP) systems [27].

Following this paradigm, we propose a data center network architecture that leverages commodity switches interconnected in a Clos topology. This topology has the nice property of providing full bisection bandwidth, something that today's architectures can only achieve at a very high cost. Further, we propose a unique routing algorithm called *two-level lookup*, that aims to balance traffic across the network.

The rest of the thesis is organized as follows. Chapter 2 gives the results of a survey regarding current best practices in data center network architectures, highlights some of their shortcomings, and gives cost figures for different cluster sizes. The details of our architecture are also discussed. Chapter 3 gives an overview of some of the most important related work. Chapter 4 discusses the implementation of the two-level routing scheme in hardware and describes an actual implementation. Finally, in Chapter 5 I give an overview of the implementation of a simulator for our architecture.

Chapter 2

Background

This chapter gives the necessary background on communication networks for data centers. Initially, I present some current facts and trends regarding data center designs. More specifically, the discussion focuses on one of the main driving forces behind data center design, which is the commoditization of the data center. I continue by describing the two different approaches in designing communication networks for large clusters and explain why non-commodity parts are currently indispensable in data center design. The discussion includes examples of actual designs along with their associated costs. The chapter concludes with a description of a data center architecture that leverages commodity routers and employs a novel routing technique called *two-level routing* [8].

2.1 Current data center network architectures

During the last few years, there has been a shift from handcrafting client/server systems out of custom components based on semi-standard infrastructure, to assembling and configuring virtual data centers out of commodity components (servers, storage systems, and networks) [1]. The drive behind this shift is mostly economic, since the commoditization of the data center significantly decreases the total cost of ownership

(TCO). This, in turn, has made it possible to build clusters that consist of hundreds of thousands of nodes, designed to meet the growing needs of many classes of applications both in the enterprise and in research labs and universities.

Although the use of commodity servers and storage systems has become standard practice in the data center, the same does not hold for the communication networks that are used to interconnect data centers. In fact, the network infrastructure is comprised—to a large extent—of non-commodity components. The reason for this is that until now it has been impossible to provide the necessary bandwidth between nodes without the use of non-commodity equipment. For this reason, most data center designs employ high-end network infrastructure that is capable to provide the necessary bandwidth. Many of the applications that run on these clusters are highly distributed and have significant intra-node communication requirements. Examples include MapReduce [15], distributed file and storage systems [21, 10], Internet services [16] and parallel scientific applications [11].

Today there are two high-level approaches for building communication networks for large-scale clusters. One approach leverages specialized hardware and communication protocols to provide a scalable, high-performance interconnect. Among the most popular interconnects that fall into this category are InfiniBand [6] and Myrinet [9]. The major drawback of such approaches is that they and are not natively compatible with applications using TCP/IP. Also, they require non-standard network interfaces at the end hosts, which makes this a rather expensive approach. An overview of such approaches is given in Chapter 3.

The second approach leverages commodity Ethernet switches and routers as the building blocks of the communication network. This approach has the advantage of providing an infrastructure that offers wide compatibility with existing applications, operating systems and hardware. Also, because of Ethernet's ubiquity, such solutions are easier to install and manage. On the downside, in such approaches the available bandwidth scales poorly with cluster size and achieving the highest levels of bandwidth incurs non-linear cost increases with cluster size. Furthermore, the Ethernet's main strength—its ubiquity—can also be viewed as its weakness, since changing its standards to adapt to growing application needs (e.g. a 100 Gb/sec Ethernet standard) is a much more timely and involved process than for more specialized interconnects. Nevertheless, Ethernet-based approaches are widely deployed in the enterprise and have gained significant share in High-Performance Computing environments (see Figure 2.1).

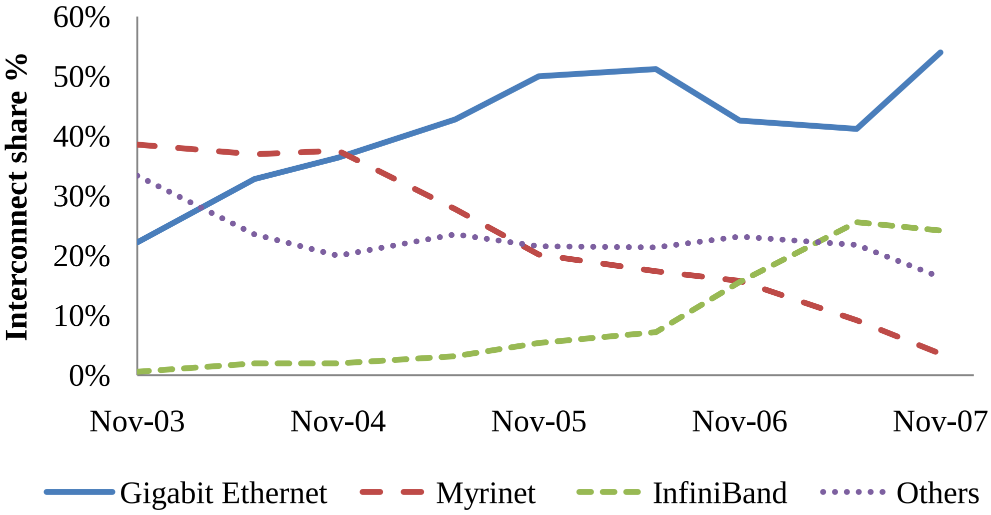


Figure 2.1: Interconnect family share in the Top500 supercomputer list [5] between 2003–2007.

As a closing remark, it is worth mentioning that recently there has been a convergence between Ethernet-based and specialized interconnects. Recent efforts try to consolidate the benefits of both Ethernet and InfiniBand in a single, unified high performance computing fabric, which is protocol-agnostic, while improving management, reliability, and security [20]. Similarly, Myricom's new Myri-10G line of products has been designed to be compatible with both Myrinet and Ethernet [2].

2.2 Overview of Ethernet-based data center architectures

As part of my research, I conducted a study to determine current best practices for data center communication networks, mainly focusing on commodity designs lever-

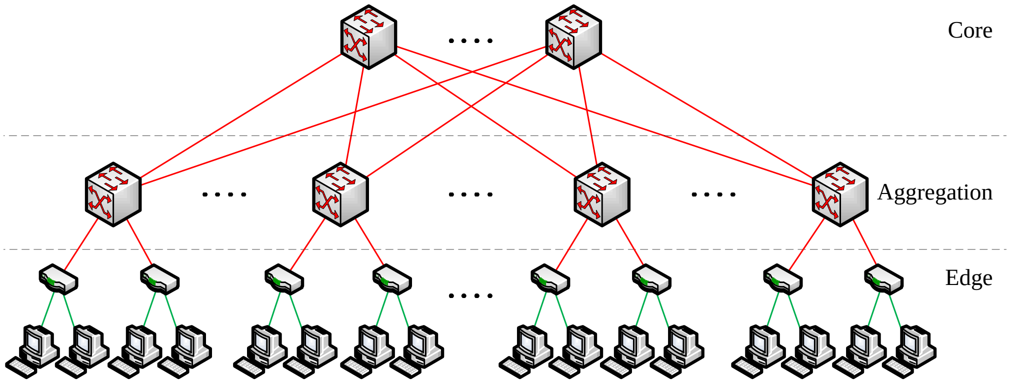


Figure 2.2: Typical layered design.

aging Ethernet and IP. Here, I give an in-depth overview of Ethernet-based data center designs; alternative interconnect options and their relation to our work are discussed in Chapter 3.

2.2.1 Topology

Most data center network designs are based on a hierarchical, layered topology. These designs have been extensively used and improved over the years and have been shown to promote scalability, performance, flexibility, resiliency, and manageability in the data center. Typical layered designs are either two- or three-tiered. For large-scale clusters, three-tiered designs are usually employed, because they can support a much larger number of hosts (typically a two-tiered design can support between 5K and 8K hosts, whereas a three-tiered design can support more than 25K hosts). Three-tiered designs also promote flexibility, scalability and lead to a lower TCO. Figure 2.2 shows a typical three-tiered design. The three layers of the data center design are the *core*, *aggregation*, and *edge* layers. Below is a brief description of the functionality of each of these three layers.

- **Core layer:** The data center core layer provides a fabric for high-speed packet switching between multiple aggregation modules. Links connecting the data center core typically use 10 GigE interfaces for supporting a high level of throughput, performance, and meeting desired oversubscription levels.

- **Aggregation layer:** The aggregation layer, with many access layer uplinks connected to it, aggregates traffic between different subnets. The aggregation switches support many 10 GigE and GigE interconnects and provide a high-speed switching fabric with a high forwarding rate. The aggregation layer usually also provides value-added services, such as firewall, server load balancing, SSH offloading, and others.

- **Edge layer:** The edge layer provides the physical level attachment to the server resources, and operates in layer 2 or layer 3 modes. The edge layer aggregates the server traffic onto (usually) 10 GigE uplinks to the aggregation layer.

2.2.2 Oversubscription

It is common in many data center designs to introduce oversubscription as a means to lower the total cost of the design. We define the term *oversubscription* to be the ratio of the aggregate bandwidth available among the end hosts to the total bisection bandwidth of a particular communication architecture. An oversubscription of 1:1 indicates that all hosts may potentially communicate with arbitrary other hosts at the full bandwidth of their network interface (e.g. 1 Gb/s for commodity Ethernet servers). An oversubscription value of 5:1 means that only 20% of available host bandwidth is available for some communication patterns. Oversubscription ratios of 2.5:1 (400 Mbps) to 8:1 (125 Mbps) are quite common in current data center designs [7]. Although data centers with oversubscription of 1:1 are possible for 1 Gb/s Ethernet, as I discuss in Section 2.2.4, the cost for such designs is almost prohibitive, even for modest-size data centers.

2.2.3 Multi-path routing

Delivering full bandwidth between arbitrary hosts in larger clusters requires a "multi-rooted" tree with multiple core switches (see Figure 2.2). This in turn requires a multi-path routing protocol because single-path routing protocols (e.g. OSPF [30]) are not able to take advantage of multiple paths in the topology and would quickly lead to highly oversubscribed links. Currently, most enterprise core switches support Equal Cost Multi-Path (ECMP) [40, 22]. ECMP permits additional links to be added between the core and access layer as required, providing a flexible method of adjusting oversubscription and bandwidth per server. Without the use of ECMP, the largest cluster that can be supported with a singly-rooted core is limited by the port capacity of the core switch. For example, using a 128-port 10 GigE switch (which is the current high-end in switching equipment), we would be able to accommodate merely 1,280 GigE end-hosts with 1:1 oversubscription. More details on the implementation of ECMP and its relation to our work are discussed in Chapter 3.

2.2.4 Cost analysis

As discussed above, cost is a significant factor in data center design and many times oversubscription is introduced as a trade-off between bandwidth requirements and cost. I conducted a brief survey to compute the rough cost (in 2008 prices) of various configurations for different number of hosts and target oversubscription ratios using current best practices.

This discussion assumes the use of two types of switches. The first, used at the edge of the architecture, is a 48-port GigE switch, with four 10 GigE uplinks. The higher levels of the hierarchy employ 128-port 10 GigE switches, which represent the current high end in both port density and bandwidth. The calculations assume a cost of $7,000 for each switch at the edge and $700,000 for 10 GigE switches in the aggregation and core layers. Any cabling or other peripheral costs are not considered in these calculations.

Minimum oversubscription designs

To achieve an oversubscription of 1:1, we need to impose the restriction that we only use 40 out of the available 48 ports in the edge switches, since each edge switch has only four 10 GigE uplinks. We next connect half of the 128 ports of the aggregation switches to the uplinks from the edge switches and the remaining 64 ports to the core switches. Therefore, each aggregation switch can accommodate $64/4 = 16$ edge switches and each core switch can accommodate two aggregation switches.

Using this design, the largest network we can build supports 23,040 hosts. The edge tier consists of 576 48-port GigE switches. The aggregation layer consists of 36 128-port 10 GigE switches, in groups of four. The core consists of an additional 18 128-port 10 GigE switches, each of which is connected to every aggregation switch with four channels per switch. Figure 2.3 plots the cost in US dollars for minimum oversubscription designs, as a function of the maximum number of supported hosts. It becomes evident that even for medium size clusters the cost becomes prohibitive. For instance, the switching hardware to interconnect 25,000 hosts with full bandwidth among all hosts comes to approximately $46 million. Since one of our design goals is to provide full bisection bandwidth and support roughly the same number of hosts, Figure 2.3 also plots the projected cost for our architecture in comparison.

Higher oversubscription designs

Simply by utilizing all 48 ports in the edge switches in any design, we can accommodate 20% more hosts, at the expense of an oversubscription of 1.2:1. The cost can be further reduced by introducing oversubscription at the higher tiers. This is done by spreading the 128 ports of the aggregation tier switches unevenly between the ports. For example, if we choose to assign 96 ports as ingress and 32 ports as egress (and hence introduce an oversubscription of 3:1), we can accommodate 23,040 hosts using only 24 switches in the aggregation layer (instead of 36) and 6 switches in the core (instead of 18). Figure 2.3 plots the rough cost for different oversubscription ratios

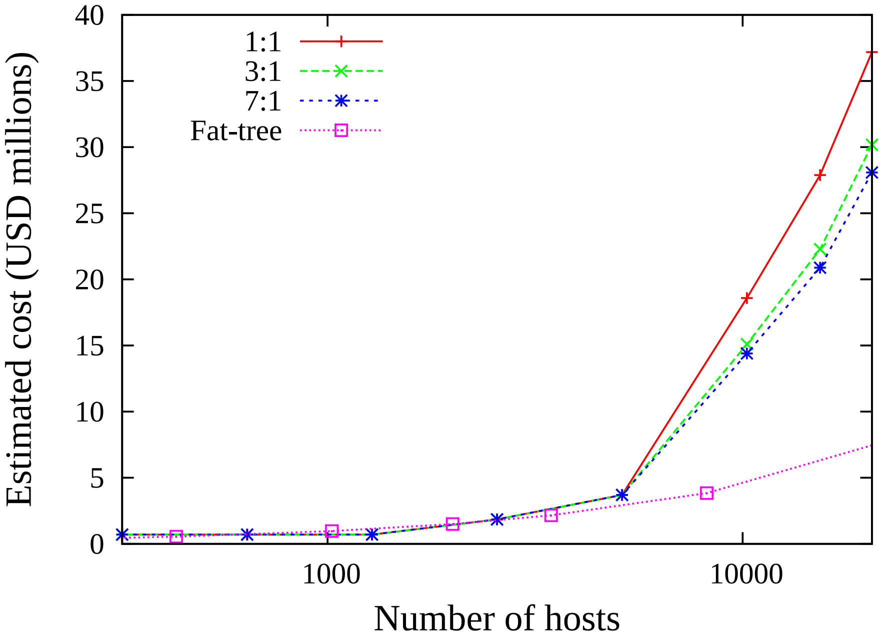


Figure 2.3: Cost for different oversubscription ratios.

commonly found in current data center designs. The cost for minimum oversubscription designs and our architecture is also plotted for comparison.

2.2.5 Historical trends

Although Figure 2.3 gives a strong hint that our architecture has a significant advantage over existing commodity interconnects in terms of cost, here I highlight a perhaps more fundamental advantage of our architecture. By conducting a survey using historical data between 2002 and today, I was able to show that our architecture is able to take full advantage of current state of the art in switching technology and can be used to build clusters with much higher capacity and bandwidth than traditional designs.

Using product announcements from various vendors of high-end switches, I tried to determine the size of the largest possible cluster using the state of the art in switching equipment at the time, both using traditional designs and our architecture. To allow for a fair comparison, all the designs had a 1:1 oversubscription requirement. Figure 2.4 and

Table 2.1: Historical trends of Ethernet data center interconnects.

	Hierarchical design			**Fat-tree**		
Year	**10 GigE**	**Hosts**	**Cost/ GigE**	**GigE**	**Hosts**	**Cost/ GigE**
2002	28-port	4,480	$25.3K	28-port	5,488	$4.5K
2004	32-port	7,680	$4.4K	48-port	27,648	$1.6K
2006	64-port	10,240	$2.1K	48-port	27,648	$1.2K
2008	128-port	20,480	$1.8K	48-port	27,648	$0.3K

Table 2.1 plot the findings of this survey. By examining these figures, we can see that there is a direct connection between the port densities of the switches and the limitation of the largest cluster possible in each year. In contrast, a 27K-host cluster would have been possible since 2004, were our architecture available in these years. Furthermore, the cost of 10 GigE switches needed to build a large cluster has been almost prohibitive until very recently. On the other hand, our architecture leverages commodity GigE switches without 10 GigE uplinks, which are required in traditional designs and drive the total cost even higher.

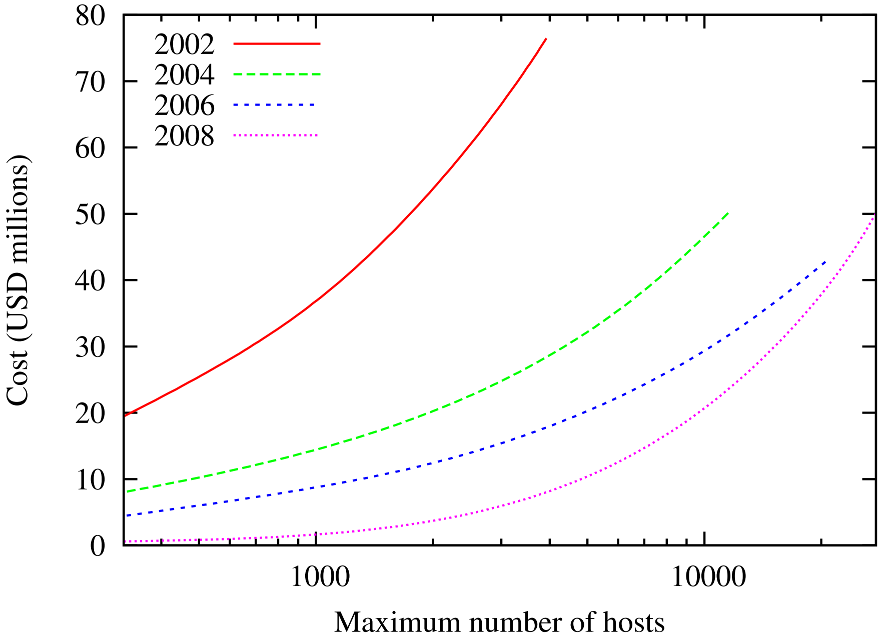


Figure 2.4: Cost for 1:1 oversubscription.

Finally, it is interesting to note that it is technically feasible to build a 27K-host cluster with 10 Gb/s bandwidth potentially available among all nodes. This potential is very relevant today, when 10 GigE switches are on the verge of becoming commodity parts; there is roughly a factor of 5 differential in price per port per bit/sec today and this differential continues to shrink. While the cost of this interconnect would likely be prohibitive in most settings, the bottom line is that it is not even possible to build such a configuration using traditional aggregation with high-end switches because today there is no product or standard for Ethernet switches faster than 10 GigE.

2.3 Clos networks

Our architecture is influenced by Clos networks. In this section I give a brief description of the architecture and properties of Clos networks, as they relate to our work.

A Clos network is a three-stage[1] switching network that was introduced in 1953 by Charles Clos in [13]. The main point of Clos's work is that it is feasible to appropriately interconnect smaller switches and deliver levels of bandwidth comparable to a single massive switch. Although the original work was geared towards the telephone network, the fundamental concept this work is still valid today, since it is prevalent in most of the current high-end clusters and supercomputer architectures. For example, Sun Microsystems recently announced a 3,456-port InfiniBand switch that consists of 720 24-port switching elements in a five-stage Clos network [4].

2.3.1 Structure and properties of Clos networks

A three-stage Clos network is defined by a triple (m, n, r). A Clos network has m middle-stage and r ingress and egress switches. Every middle-stage switch is connected to each of the r ingress and egress switches. Figure 2.5 shows an instance of a $(5, 4, 3)$

[1] Clos networks with any odd number of stages can be derived recursively from the three-stage Clos network by replacing the switches of the middle stage with three-stage Clos networks [14]

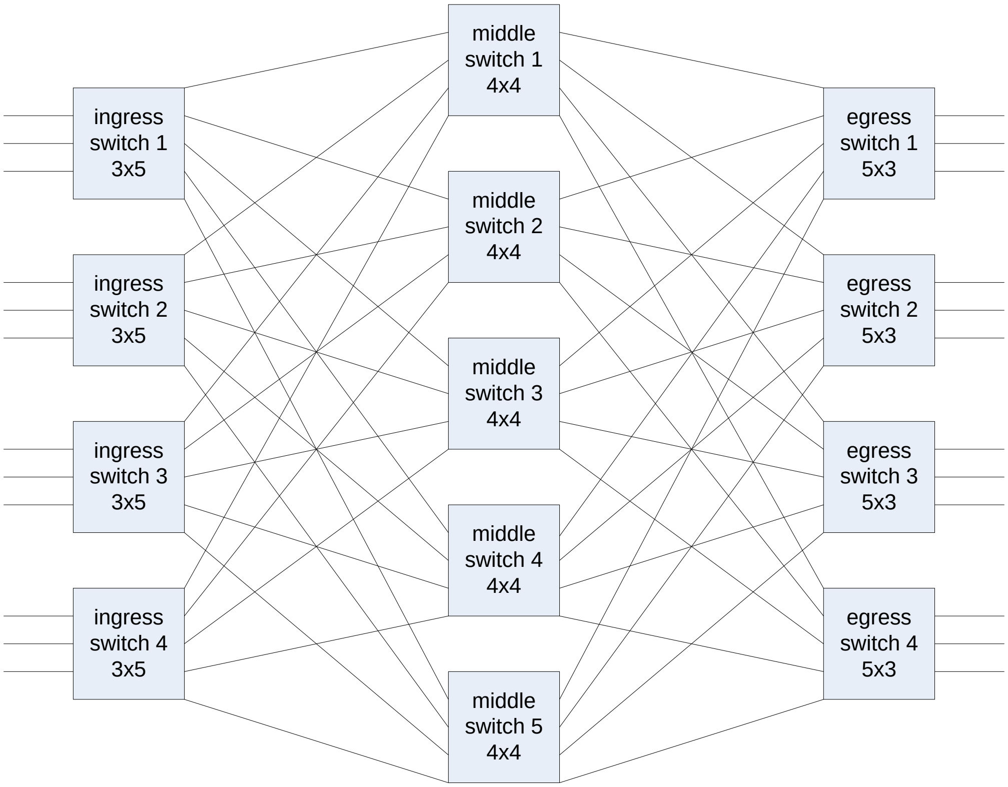


Figure 2.5: A (5,3,4) Clos network. There are $m = 5$ $r \times r$ middle switches, $r = 4$ $n \times m$ ingress switches and $r = 4$ $m \times n$ egress switches.

Clos network. This network can support 12 ($= 4 \times 3$) simultaneous connections in a circuit-switched network.

One of the reasons that makes Clos networks a very attractive interconnect is their nonblocking properties. A circuit-switched network is said to be *nonblocking* when a dedicated path from an input port on an ingress switch to a certain output port on an egress switch can be selected without conflicts (e.g. shared channels). Depending on the values of m and n, Clos networks have different non-blocking properties. A *strictly non-blocking* network can be set up incrementally, meaning that any free input port can be connected to any free output port without the need to rearrange existing calls. A (*m*, *n*, *r*) Clos network is strictly non-blocking iff $m \geq 2n - 1$. In contrast, in a *rearrangeable* network setting up calls incrementally may require rearranging some of the existing calls. A Clos network is rearrangeable iff $m \geq n$. Strictly non-blocking Clos networks are nearly twice as expensive, requiring $\frac{2n-1}{n}$ times as many middle switches as a corresponding rearrangeable network. For this reason, a rearrangeable network is usually preferred.

Although the term *non-blocking* is used to describe circuit-switched networks, the term has been overloaded and has widely used to describe packet-switched networks[2]. Since in packet-switched networks channels are shared and there is no notion of call setup, the definition of the non-blocking property in such networks is not relevant and has a different meaning. We say that a packet-switched network is *non-blocking* if (a) the maximum load for each channel does not exceed the channel's bandwidth and (b) there is no coupled resource allocations between flows that share the same channel [14].

2.3.2 Fat-trees

Fat-trees are a special instance of Clos networks, which were introduced as an efficient interconnection network for supercomputers [28]. A fat-tree can be viewed as a folded multi-stage Clos network. Unlike ordinary tree data structures, a fat-tree has multiple roots and hence has multiple paths between nodes Most fat-trees are built using identical switches in all stages. This ensures that fat-trees are guaranteed to be rearrangeable (see Section 2.3.1). For these reasons, fat-trees make for a very attractive interconnect topology and as I discuss in Section 3.1, fat-trees are widely used in HPC clusters as well as internally in switches.

The organization of a fat-tree is similar to the layered design discussed in Section 2.2.1. An example of the simplest non-trivial fat-tree that is built out of 4-port switches is shown in Figure 2.6. In general, a k-ary fat-tree is built using switches with k bidirectional ports. The lower two tiers are grouped into k *pods*, with each pod consisting of $k/2$ switches in the lower and $k/2$ switches in the upper tier. Each lower tier switch is connected to all of the $k/2$ upper tier switches in its pod, as well as to $k/2$ end nodes. At the root there are $(k/2)^2$ *core* switches. Each core switch is connected to one

[2] Dally and Towles [14] suggest the term *non-interfering* for packet-switched networks as the equivalent of the term *non-blocking* in circuit-switched networks.

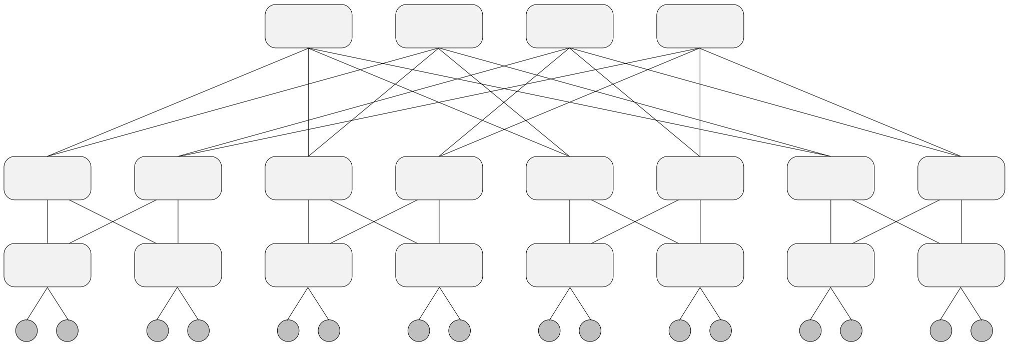


Figure 2.6: The simplest non-trivial fat-tree using 4-port switches.

switch in every pod. As a result, in a fat-tree there are $k^2/4$ equal-cost paths between any pair of end nodes in distinct pods. This is because there are $(k/2) \cdot (k/2)$ possible paths from the source to any core switch and a single path from a core switch to the destination.

2.4 A commodity data center network architecture

As discussed in Section 2.1 and 2.2, current data center network architectures leverage mostly non-commodity parts and are heavily oversubscribed. Recently, a novel data center network architecture was introduced [8], which leverages strictly commodity switches organized in a fat-tree and provides full bisection bandwidth for tens of thousands of hosts. This section gives a brief overview of the architecture, focusing on the details that are necessary for the discussion in Section 4 and 5. For more details, the reader is referred to the original paper.

2.4.1 Topology

The network is organized in a fat-tree topology as follows (see Figure 2.7). The network is built using identical k-port switches. There are k pods, each containing two layers of $k/2$ switches. Each lower-level pod switch is directly connected to $k/2$ hosts. Each of the remaining $k/2$ ports is connected to $k/2$ of the k ports in upper-level pod

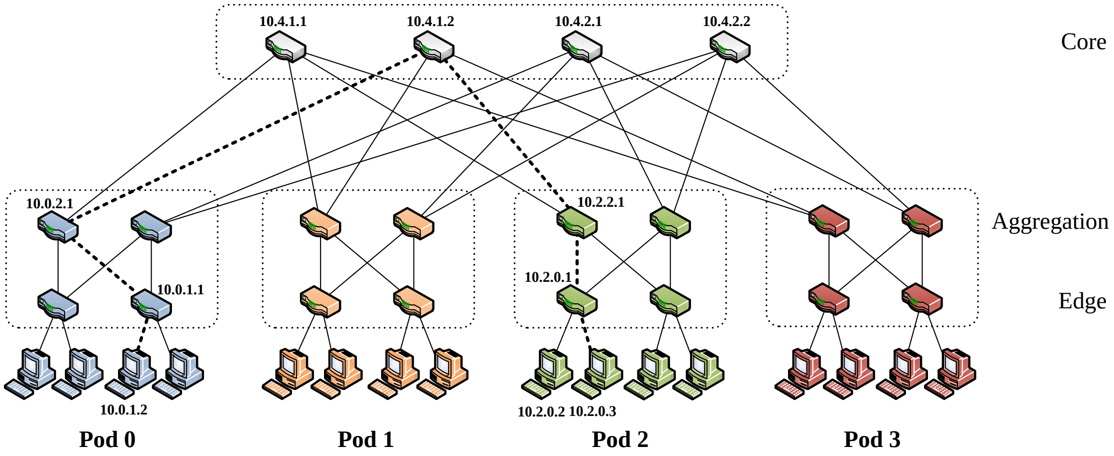


Figure 2.7: Example of address allocation in a fat-tree.

switches. There are $(k/2)^2$ core switches arranged in a square grid. Each core switch has one port connected to one of the pods. There are $k * (k/2)^2$ links out of the core grid, and $k * (k/2)(k/2)$ links coming out of all the pods. The i^{th} port of any core switch is connected to pod i such that consecutive ports in the aggregation layer of each pod switch are connected to core switches on $k/2$ strides. All told, a fully populated network consists of $5k^2/4$ switches and supports $k^3/4$ end hosts.

2.4.2 Addressing

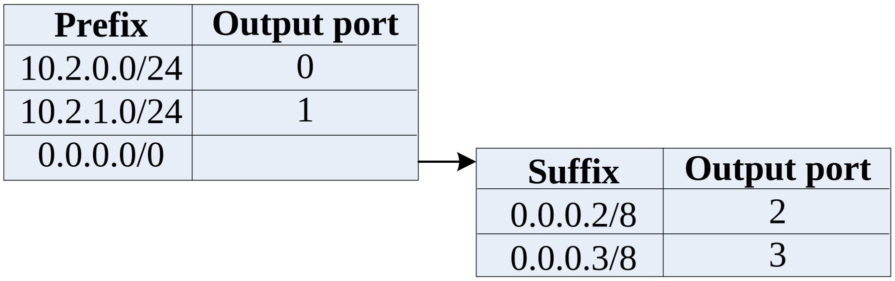
The architecture employs a systematic addressing scheme, such that the IP address of a host or router encodes its location in the fat-tree. This information is a key part of the two-level routing algorithm, as discussed below.

All the IP addresses in the cluster are within the private $10.0.0.0/8$ block. The pod switches are given addresses of the form $10.pod.switch.1$, where *pod* denotes the pod number (in $[0, k-1]$), and *switch* denotes the position of that switch in the pod (in $[0, k-1]$, starting from left to right, bottom to top). The core switches are given addresses of the form $10.k.j.i$, where j and i denote that switch's coordinates in the $(k/2)^2$ core switch grid (each in $[1, (k/2)]$, starting from top-left). The address of a host then follows from the pod switch it is connected to; hosts are given addresses of the form: $10.pod.switch.ID$, where *ID* is the host position in that subnet (in $[2, k/2+1]$, starting from left to right). Therefore, each lower-level switch is responsible for a $/24$ subnet of $k/2$ hosts.

2.4.3 Two-level lookups

One of the key concepts in the architecture is that of the *two-level lookup*. The two-level lookup is a modification to the traditional longest matching prefix algorithm that is used by IP routers today. The two-level lookup takes advantage of the large fan-out of the fat-tree topology and is able to provide full bisection bandwidth.

The two-level lookup requires a slight modification to the structure of the routing tables, so that a table may point to *secondary* tables. Figure 2.8 shows an example of a modified routing table. Each entry in the main routing table may potentially have an additional pointer to a small *secondary* table of $(prefix, port)$ entries. Such entries are called *non-terminating entries*. In contrast, *terminating entries* are identical to regular routing table entries. While a non-terminating entry may point to only one secondary table, a secondary table may be pointed to by more than one non-terminating entries. Primary table entries are regular prefix entries of the form $1^m 0^{32-m}$, while secondary table entries are *suffix* entries of the form $0^{32-n}1^n$.



Prefix	Output port
10.2.0.0/24	0
10.2.1.0/24	1
0.0.0.0/0	

Suffix	Output port
0.0.0.2/8	2
0.0.0.3/8	3

Figure 2.8: An example of a two-level routing table.

2.4.4 Routing

The routing algorithm using the two-level table follows from the table structure. For each packet, the longest matching prefix algorithm is initially run on the primary table. If the hit is a terminating entry, then the packet is forwarded to the port designated by the entry, just as with the regular IP routing mechanism. However, if the hit is a non-terminating entry, then the secondary table pointed by the entry is searched as well. In this case, the algorithm looks for the longest matching *suffix* in the secondary table pointed by the non-terminating entry. The packet is then forwarded to the port indicated by that entry.

Chapter 3

Related work

3.1 Related fat-tree architectures

Interconnection networks based on fat-tree topologies are very common in large-scale clusters and supercomputers. Here I give an overview of some of the most representative architectures and discuss their relation to our work.

3.1.1 Thinking Machines CM-5

The CM-5 [27] is a massively parallel computer system that was introduced by Thinking Machines Corp. in 1991. The data network in the CM-5 is a fat-tree packet-switched network that is able to interconnect between 32 and 16,384 processing nodes. The network is logically divided into "user partitions", which essentially correspond to subtrees. Traffic local to a partition is contained within the partition's subtree and thus relieves higher levels of the tree from carrying unnecessary traffic. The routing function used in the CM-5 is randomized: at the hops between the source node and the root of the tree, routers make their forwarding decisions by pseudo-randomly selecting between available uplinks. Once a message has reached the root, there is a single deterministic path to the destination node. The introduction of randomness in routing aims to perform load balancing across links. This has been shown to perform well for regular communication patterns commonly found in massively parallel processing [26].

3.1.2 Myrinet

Myrinet [9] is a high-performance interconnect for computer clusters designed by Myricom. Myrinet was designed as an alternative to Ethernet and provides low-latency communication by implementing proprietary protocols that run in user-space and override expensive system calls. The system architecture employs sophisticated network interface cards (NICs) on the end hosts and simple low-overhead switches. Myricom offers switches with 8–512 ports, which can be used to build networks up to 8,192 hosts using a fat-tree topology. Most of the power of Myrinet is found in the NICs. The NICs on the hosts predetermine routes and the switches simply forward packets based on routing information in the packet header using cut-through switching. The sources monitor contention on the route and change the path that flows take by taking advantage of the multiple paths in the fat-tree. Current Myrinet implementations allow for 8-way multi-path routing. Fault tolerance is also dealt with by means of source routing by having the sources identify high delays in receiving ACKs. A problem with Myrinet's routing scheme is that it does not guarantee in-order packet delivery, which is known to be detrimental to TCP performance. NICs at the destination host are responsible for reordering packets.

3.1.3 InfiniBand

InfiniBand is a switched fabric interconnect that has been developed to provide high-performance I/O between nodes in a computer cluster. Current InfiniBand links offer a peak bandwidth of 96 Gb/s with an end-to-end latency under $3\mu s$. With the use of dedicated hardware (called Channel Adapters) that provides an InfiniBand interface that bypasses the O/S and the CPU, and a proprietary protocol stack, InfiniBand can provide high-throughput and low-latency data transfers between nodes. The InfiniBand protocol stack is analogous to layers 1–4 of the OSI protocol stack and provides reliable, in-order transmission of packets.

Hosts are attached to InfiniBand switches and are organized in subnets. The switches forward packets in the same subnet using the destination local ID (LID) and are also responsible for providing QoS and flow control. InfiniBand switches with as many as 3,456 ports have been introduced [4], which are internally comprised of many switching elements interconnected in a 5-stage Clos network (fat-tree). Clusters up to approximately 13K end nodes can be build with the use of multiple such switches interconnected in a fat-tree topology. Theoretically, even larger clusters can be constructed by interconnecting many subnets with the use of InfiniBand routers. Despite the fact that the InfiniBand standard defines a router, which uses IPv6 to forward packets, there are currently no commercially available InfiniBand routers.

The InfiniBand subnet manager performs management tasks and ensures the correct operation of the network. The subnet manager configures forwarding tables on attached switches and assigns LIDs to end nodes and switches. Because the subnet manager is a single point of failure, the InfiniBand architecture allows for multiple redundant subnet managers that provide a graceful failover mechanism.

InfiniBand can support Ethernet by encapsulating Ethernet packets in InfiniBand payload. IP over InfiniBand (IPoIB) [12, 25, 23, 24] enables IP communications using either IPv4 or IPv6.

3.1.4 QsNet

QsNet is a cluster interconnect system designed by Quadrics. It consists of a programmable NIC and a high-bandwidth, low-latency switch. Switches can be connected in a fat-tree topology to form larger networks. QsNet employs source routing, where the NIC attaches routing tags to packets that can specify a link or a group of links. Fault detection is done in hardware by both the NICs and the switches; faulty links and/or switches are identified and isolated, and routing tables at the sources are appropriately reconfigured. QsNet also provides a software libraries that, similar to

Myrinet, implement communication protocols in user-space.

3.1.5 NUMAlink

NUMAlink is a very high performance interconnect designed by Silicon Graphics Inc. and has been deployed in their Origin and Altix computer systems. NUMAlink is deployed in a dual fat-tree topology that provides performance and fault tolerance. Performance is achieved by using very small packet sizes and by reducing forwarding delays by employing circuit-switching. The dual fat-tree topology provides fault tolerance by enabling one plane to operate without the other and disabling defective links or nodes (called Bricks) on the fly. However, the scalability of the architecture is limited to a maximum of 2,048 nodes for NUMAlink version 4.

3.2 Multi-path routing protocols

Multi-path routing protocols enable the discovery and use of multiple best paths to a single destination, as an effort to perform load balancing of network traffic over redundant paths. Equal-cost multi-path (ECMP) [40] is one such routing protocol, in which a router stores many best paths (in terms of some common cost metric) to a specific destination and makes local, per-hop forwarding decisions.

There are two classes of ECMP forwarding mechanisms. The first performs load balancing by doing per-packet multi-path routing, usually by using a round-robin or randomized algorithm. This approach is now deprecated, because such algorithms are disruptive, that is, they may cause packets that belong to a specific flow to take different paths. This in turn can lead to packet reordering, which is potentially detrimental for TCP. Packet fragmentation is also possible, since multiple paths may have different MTUs and thus make path MTU discovering useless. For these reasons, per-packet ECMP routing is rarely used in practice.

Instead, a common technique uses a hash of flow-related data and splits flows among a set of ports [22]. This technique deterministically sends all packets that belong to the same flow to the same output port, while balancing multiple flows over multiple paths in general. However, this technique does not account for flow bandwidth in making allocation decisions, which can quickly lead to oversubscription even for relatively simple communication patterns. Further, current ECMP implementations limit the multiplicity of paths to 8–16, which is often less path diversity than required to deliver high bisection bandwidth. The reason for this is that the number of routing table entries grows exponentially with the number of paths considered, which can increase the cost of the switch.

In addition to its shortcomings and difficulties in deployment, there are many situations where the network not only does not seem to benefit from ECMP routing but ECMP can also have a negative effect. For example, it is very common that multiple paths to a certain destination end up converging into a single path along the way. This way, the added complexity and cost of ECMP does not seem to yield performance improvements. In other cases, ECMP can adversely affect network operation and performance. This is common in systems where the physical topology differs from the logical topology (e.g. VLANs).

Chapter 4

Two-level lookup implementation

This chapter describes possible hardware implementations of the two-level lookup. I initially give the necessary background on the prefix lookup problem as well as the two classes of techniques that are used in practice. The rest of the chapter deals with how to implement the two-level lookup using hardware techniques and describes an actual implementation as well as experimental results.

4.1 Prefix lookups

The implementation of address lookups in routers is a challenging problem and can greatly affect the router's performance. The main difficulty that arises with prefix lookups is due to IP's classless inter-domain routing (CIDR) scheme [36] that allows for arbitrary prefix lengths. In this case, the address lookup involves performing a longest prefix matching (LPM) algorithm on the routing table, in order to find the table entry with the longest prefix matching with the destination IP address. Effectively, this is a two-dimensional search algorithm, searching both in values and length. In the case of IPv4, this means that the LPM algorithm must search for an exact matching prefix for all prefixes with length $i = 1, 2, ..., 32$. This task becomes even more difficult with the increasing number of routing table entries, which are in the order of hundreds of thousands. Finally, the very high data rates of the links imposes a very tight upper bound on the lookup time per packet (e.g. 32 nsec for a 10 Gbps link). Here, I give an overview on current best practices for implementing prefix lookups in forwarding engines. There are two high-level approaches to this problem. The first leverages algorithmic techniques with sophisticated data structures, while the second leverages specialized hardware.

4.1.1 Algorithmic techniques

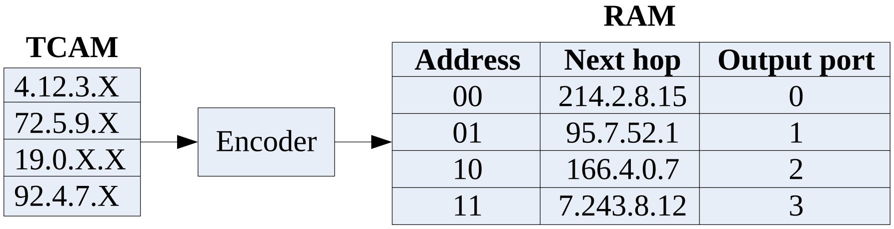
Algorithmic techniques use ordinary RAM to store the routing table and perform lookups. In such schemes, the main performance metric is the number of memory accesses per lookup. Algorithmic techniques employ data structures that guarantee a small upper bound on the number of memory accesses per packet. Most notable algorithmic techniques use some variant of a trie data structure [39, 17, 18]. A more detailed overview of algorithmic techniques can be found in [37].

4.1.2 Hardware techniques

One of the most common hardware solutions for performing address lookups is Content-Addressable Memory (CAM) [34]. A CAM is a fully associative memory that can store binary information and is capable of performing parallel searches among all its entries. CAMs are composed of conventional semiconductor memory (usually SRAM) with added comparison circuitry that enables a search operation to complete within a single clock cycle.

A Ternary CAM (TCAM) is a special kind of CAM that is able to store *don't care* bits in addition to 0s and 1s that, as the name implies, match both 0 and 1. This makes a TCAM suitable for storing variable length prefixes, such as the ones found in routing tables. Compared to software-based techniques, address lookup engines that are based on TCAMs provide a constant lookup time (one cycle per lookup) and are a much more straightforward solution. On the other hand, TCAMs have rather low storage density (roughly 2-3x bits/area less than SRAM), they are very power hungry (roughly 150x more power/bit than SRAM), and incur a significant cost (roughly 30x more cost/bit than SRAM). Current research in CAMs focuses on reducing cost and power consumption [33, 31, 32].

An example a address lookup engine using a TCAM is shown in Figure 4.1. Prefix lookups are done in a two-step fashion. First, the lookup engine does a lookup on the TCAM to find the longest matching prefix. This lookup is done in parallel and completes in a single cycle, in approximately 5–20 nsec. The match address of the TCAM is in turn used to index a RAM (usually SRAM) that stores the associated data (i.e. next hop IP address and output interface). Because in many cases there may be multiple matches in the TCAM, the search result is encoded before indexing the RAM. Usually, the encoder outputs the numerically smallest matching address. For this reason, the order in which TCAM entries are stored is crucial. This has a direct effect in the time complexity of updating the TCAM, where an insertion can take $O(M)$ time in the worst case, where M is the number of prefixes stored in the TCAM. Several methods have been proposed which achieve update times comparable to algorithmic methods [38, 35].



TCAM
4.12.3.X
72.5.9.X
19.0.X.X
92.4.7.X

Address	Next hop	Output port
00	214.2.8.15	0
01	95.7.52.1	1
10	166.4.0.7	2
11	7.243.8.12	3

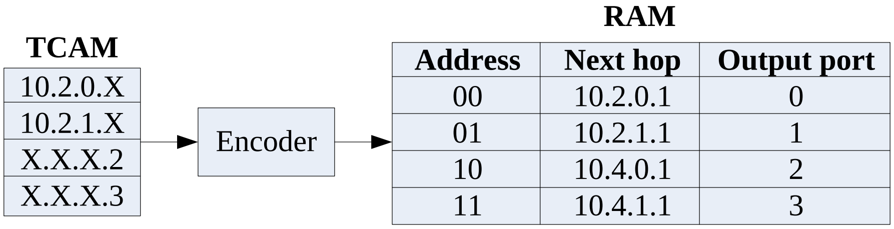
Figure 4.1: An example of a traditional routing table implemented with a TCAM. Prefixes are stored using 0s and 1s and the least significant bits are using *don't care* bits (X).

4.2 Implementation of the two-level lookup

After having given the necessary background on address lookups, I now describe a possible implementation of the two-level lookup scheme, as described in Chapter 2. The discussion assumes that all IP addresses follow the addressing scheme described in Chapter 2.

4.2.1 TCAM implementation

The two-level lookup can be implemented in a straightforward manner using a TCAM, as described in Section 4.1.2. Although such CAM-based solutions have their shortcomings, most of them are of little or no impact when used in this architecture. First, the size of the routing tables is very small ($O(k)$ for a k-port switch), which means that the size of the CAM is also very small. More specifically, the TCAM that is needed for the two-level lookup is $\log k$ bits deep and 32 bits wide. This is currently on the low-end of TCAM specifications and hence, should not impact the total cost of the solution. Second, the routing tables are designed to be mostly static and, therefore, the TCAM does not suffer from delays due to routing table updates.



TCAM

10.2.0.X
10.2.1.X
X.X.X.2
X.X.X.3

RAM

Address	Next hop	Output port
00	10.2.0.1	0
01	10.2.1.1	1
10	10.4.0.1	2
11	10.4.1.1	3

Figure 4.2: Example of a two-level routing table implemented with a TCAM.

The proposed implementation of the two-level lookup engine is shown in Figure 4.2. A TCAM is used to store address prefixes and suffixes, which in turn indexes a RAM that stores the IP address of the next hop and the output port. Left-handed (prefix) entries are stored in numerically smaller addresses than right-handed (suffix) entries. The output of the TCAM is encoded in such a way that the entry with the numerically smallest matching address is output. This satisfies the semantics of the two-level lookup: when the destination IP address of a packet matches both a left-handed and a right-handed entry, then the left-handed entry is chosen. For example, using the routing table in Figure 4.2, a packet with destination IP address $10.2.0.3$ matches both the left-handed entry $10.2.0.X$ and the right-handed entry $X.X.X.3$. The packet is correctly forwarded on port 0. However, a packet with destination IP address $10.3.1.2$ matches only the right-handed entry $X.X.X.2$ and is forwarded on port 2.

4.2.2 Alternative implementation

The implementation presented above assumes the presence of a TCAM in the router. Although this is the most straightforward implementation, it is not the only possible. An alternative implementation could take advantage of the fact that in the addressing scheme the prefixes have a fixed length, something which can greatly simplify the lookups. At each level of the fat-tree, the prefixes (and suffixes) in the routing tables have a specific length and therefore an exact prefix matching would be sufficient. Just like the CAM implementation presented above, this scheme also has constant lookup time.

A sketch of this design is as follows. Switches store either one (core and lower pod switches) or two tables (upper pod switches). The lookup engine uses some of the bits in the destination IP address of each packet to index these tables and, with some extra logic, it selects the correct output. Recall from Chapter 2 that host IP addresses are of the form $10.pod.subnet.hostID$.

Core switches store a single table with k prefix entries. Packets are forwarded based on their pod in their destination IP address. Hence, the lookup engine uses bits 23:16 to index its lookup table. In a similar fashion, lower pod switches also store a single table but with $k/2$ suffix entries. In this case, packets are forwarded based on their $hostID$ in their destination IP address. Here, bits 7:0 are used to index the lookup table.

The process is similar, albeit a little more involved, for upper-level pod switches. In this case, there is both a primary and a secondary table. The former is indexed with bits 15:8 of the destination IP address ($subnet$) and the latter with bits 7:0 ($host$). Since both lookups will produce a hit, the lookup engine selects between them using the pod part of the packet's destination address: if the packet is destined for this pod, then the packet is forwarded on the port specified by primary table. Otherwise, it is forwarded on the port specified by the secondary table.

4.3 Implementation on the NetFPGA platform

The above presented implementation of the two-level lookup is very straightforward to implement on a router with a TCAM-based lookup engine. One such router is the one that comes with the NetFPGA [29]. The NetFPGA is a platform for network research, developed at Stanford University. It features a Xilinx FPGA, four GigE ports, and parallel banks of SRAM and DRAM, that allow for the implementation of networking hardware such as IP routers and switches using industry-standard tools (e.g. Verilog).

Among other usage modes, the NetFPGA can be programmed using open-source software provided with the hardware to operate as a hardware-accelerated IPv4 router, capable of forwarding packets at wire speed. The lookup engine uses a TCAM (implemented on the FPGA) to perform lookups, which allows us to program it with a two-level lookup table.

The implementation of the lookup engine on the NetFPGA follows the design discussed in Section 4.2.1. The TCAM allows for multiple matches on IP addresses and outputs the one that is in numerically smaller address. This way, in order to program the router to implement the two-level lookup, all left-handed prefixes should be placed in the logically higher part of the routing table.

The NetFPGA router can be programmed to behave as a 4-port two-level lookup router using a Perl script that interacts directly with the NetFPGA. The source code of a sample script can be found in Appendix B.

4.4 Evaluation

The implementation of the two-level lookup on the NetFPGA router should not affect its performance in comparison to regular longest matching prefix IP forwarding. This is because the only modification necessary is a specific programming of the routing table. The TCAM has constant lookup time for any kind of entry and should not have different lookup times for prefix and suffix entries. The evaluation process tried to verify that the two-level lookup has no overhead over the traditional longest prefix matching mechanism.

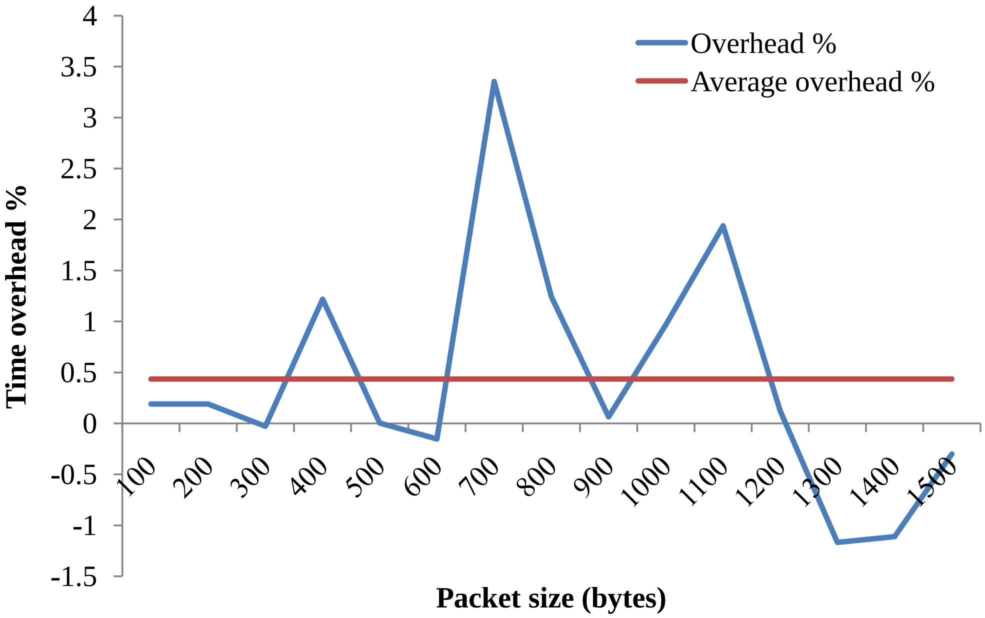


Figure 4.3: Average overhead of the two level lookup over traditional the IP lookup mechanism for different packet lengths.

The experimental setup was the following. A PC was configured with the NetFPGA board that had the reference IPv4 router bitfile programmed on its FPGA. The PC also had two Gigabit Ethernet network interface cards, both connected to the

NetFPGA. The routing table was programmed such that it reflects that of an upper-level pod switch. A simple Perl script was used to generate IP traffic with source and destination IP addresses, such that one interface sends traffic to the other interface via the NetFPGA router[1]. The traffic consisted of a fixed number of packets (10,000) with a packet size between 100 and 1,500 bytes in increments of 100 bytes. For each increment of the packet size, the experiment was run five times. Using a high-precision timer, I measured the time between the first packet was sent and after the last packet was sent, both for the prefix-matching lookup and for the two-level lookup. Figure 4.3 the relative percentage time overhead that the two-level table imposes over the traditional lookup mechanism for each packet size, as well as the average overhead. It is quite clear that the overhead over the traditional IP lookup mechanism is almost negligible (roughly 0.4% on average).

[1] That is, one interface was designated as the source and the other as the sink.

Chapter 5

Simulator implementation

This chapter describes the details of a simulator implementation, which builds upon the BookSim simulator [14]. I first give the motivation to build a simulator. I then give a brief overview of BookSim and discuss why this was our simulator of choice. Further, I present the modifications that were required to simulate our architecture using BookSim. Finally, I present some experimental performance results for different network sizes and traffic patterns.

5.1 Motivation

In order to measure the performance of our architecture in a larger scale, we resorted to simulation, simply because the number of hosts and switches in a fat-tree grows exponentially with network size. Recall from Chapter 2, that in a k-ary fat-tree there are $k^3/4$ hosts and $3k^2/4$ routers. This essentially prohibits us from conducting experiments using real end hosts and routers, even for very small values of k. For this reason, we had to resort to simulation, in order to gain helpful insight of how our system performs for larger network sizes.

Our choice was to use an already available simulator, which we could modify to simulate a fat-tree topology using the two-level routing scheme. Our simulator of choice was BookSim, a simulator provided as a complement to [14]. This simulator was chosen mostly because it provides an API that allows for the implementation of any type of interconnect topology and routing algorithm and gives full and very low-level control of the configuration of the network. Further, BookSim provides a very low-level notion of a network and is agnostic of network and transport layer protocols. Although our architecture is intended to work under the TCP/IP model, such low-level measurements would give us some notion of the bare-bones performance of the network.

5.2 The BookSim simulator

BookSim is free, open-source software and was developed at the Concurrent VLSI Architecture Group at Stanford [3]. Because most of the simulator's components are designed to be modular, it becomes easy to add new features such as new routing algorithms, topologies, or router microarchitectures, without having to redesign the code. The simulator itself is written in C++ and the modules that comprise the simulator are implemented in separate classes, forming a straightforward class hierarchy. Below I give a more detailed description on the implementation of certain modules that relate to the modifications needed to implement our architecture in BookSim. The actual source code that implements our architecture in BookSim can be found in Appendix A.

5.2.1 Routers and channels

Router and channel objects are the building blocks of the simulator's view of the network. Each router has a number of inputs and outputs, to which connect unidirectional input and output channels, respectively. Each router is also assigned a unique ID. A router object does not define a specific routing function but the simulator comes with a variety of implemented routing functions that are used in common by routers. In addition to input and output channels, BookSim also defines inject and eject channels, which play the role of sources and sinks in the network, respectively. Inject and eject channels are assigned a unique ID on creation, which serves are the source and destination address in each packet. In some sense, this is the equivalent of an IP address in the simulator, which, however, contains only identity and no location information.

5.2.2 Topology

Every network topology is implemented as a separate class. BookSim includes an implementation of some of the most common network topologies such as the butterfly and the torus. The main functionality of a topology class is in its constructor, which creates instances of routers and channels and assigns them in such a way such that they reflect the underlying topology.

5.2.3 Routing functions

Routing in the simulator is assumed to follow a global routing function common to all routers in a certain topology instance. BookSim implements some common deterministic and randomized routing functions, such as dimension-order routing and Valiant's randomized routing algorithm [41]. No tabular routing functions are implemented.

5.3 Implementation of the fat-tree in BookSim

The implementation of the fat-tree architecture in BookSim was fairly straightforward and it involved the following steps. First, I needed to modify the `router` class to include a two-level lookup table. Second, I needed to implement an additional routing function that implements the two-level routing. Third, I needed to implement a `fattree` class that implements the fat-tree topology.

5.3.1 The `fattree` class

In order to implement the fat-tree topology, I needed to implement a new topology class that builds a k-ary fat-tree. Because in BookSim router ports and channels are unidirectional, the `fattree` class builds a k-ary fat-tree using routers with $2k$ ports (k input and k output). Routers are assigned an ID ranging from 0 to $k^2/2 - 1$ for lower pod routers and from $k^2/2$ to $k^2 - 1$ for upper pod routers. Core routers are assigned numbers from k^2 to $5k^2/4$. Channels are logically grouped in pairs, each pair emulating a bidirectional channel. Inject and eject channels are each given IDs ranging from 0 to $k^3/4 - 1$.

5.3.2 Modifying the `router` class

The `router` class required a modification in order to implement our architecture. Each router object stores a two-level lookup table and implements a member function that populates the table. The member function `populateTable()` populates the table using the algorithms in [8]. Because the two-level lookup scheme greatly relies on the IP addressing scheme discussed in Chapter 2, I implemented a utility function that translates a router or inject/eject channel ID into its equivalent IP address in the fat-tree.

5.3.3 The `twolevel` routing function

Finally, I needed to implement a routing function that follows the semantics of the two-level lookup. The routing function follows the alternative implementation that uses exact prefix matching described in Section 4.2.2. Each router object stores as many tables as the different prefix/suffix lengths it stores. For example, an upper-level pod router would have two tables (one for primary $/24$ entries and one for suffix $/8$ entries), whereas a core router would have a single table with primary $/16$ entries. Tables are directly addressed by a 1:1 transformation of its IP address. The primary table is checked first and in case of a hit, the routing function returns the output port.

Otherwise, in case of pod switches, the secondary table is checked.

5.4 Experiments

Using the simulator I was able to conduct a number of experiments in order to observe the behavior of our architecture for larger network sizes. This section describes the benchmark suite I used in these experiments and gives the results of the experiments.

5.4.1 Experimental setup

All experiments were run on a single machine with a dual-core Intel Pentium CPU at 1.60GHz, with 1024KB cache and 512MB of RAM, running GNU/Linux 2.6.22. The simulator code was compiled with GCC 3.3[1], with the third level of code optimization enabled. In all experiments, the simulator was the single running process in the system, allowing an uninterrupted execution. The experiments were run for values of $k = 4, 8, 16, 20, 24$. Due to memory limitations, I was not able to run experiments for $k > 24$, although nothing prevents the simulator to be run for larger values of k on machines with more RAM. Table 5.1 gives run-time memory measurements for different values of k.

Table 5.1: Memory requirements of the simulator for different values of k.

k	**Memory requirements**
4	6KB
8	20KB
16	72MB
20	295MB
24	530MB
32	1035MB

[1] Because a large part of the core code of the simulator uses some C++ expressions that are not ISO C++ compliant and have been incorporated in later versions of the compiler, currently the code does not compile with versions of GCC later than 3.3.

5.4.2 Benchmark suite

In my experiments I used a variety of traffic patterns, in order to measure the performance of the system under different circumstances. To describe these patterns, I use the following notation: s_i (d_i) denotes the i^{th} bit of the source (destination) address[2]. The bit length of an address is $b = \log N$, where N is the number of nodes in the network.

- *Uniform*. Each source sends an equal amount of traffic to each destination.
- *Random permutation*. A fixed permutation traffic pattern is chosen uniformly at random from the set of all permutations. The random generator uses a fixed seed for all experiments.
- *Bit reverse*. $d_i = s_{b-i-1}$. This traffic pattern requires that the number of hosts is a power of 2.
- *Bit complement*. $d_i = \neg s_i$. This traffic pattern requires that the number of hosts is a power of 2.
- *Inter-pod incoming*. Multiple pods send to different hosts in the same pod, and all happen to choose the same core switch. That core switchs link to the destination pod will be oversubscribed. The worst-case local oversubscription ratio for this case is (k–1):1.
- *Same ID outgoing*. Hosts in the same subnet send to different hosts elsewhere in the network such that the destination hosts have the same host ID byte. Static routing techniques force them to take the same outgoing upward port. The worst-case ratio for this case is (k/2):1.

[2] Here address refers to an end-host's ID in $[0, 1, \ldots, k^3/4)$.

5.4.3 Experimental results

This section gives the results of the experiments. The performance metric used is the average throughput of the system. This was calculated as follows. The simulator injects traffic with a rate of x packets per simulation cycle and at the end reports the overall average accepted packet rate y across all routers. The average throughput is simply x/y. Table 5.2 summarizes the average throughput for the aforementioned traffic patterns.

Table 5.2: Average throughput for the different traffic patterns.

k	**Uniform**	**Random permutation**	**Bit complement**	**Bit reverse**	**Inter-pod incoming**	**Same ID outgoing**
4	93.833%	94.134%	99.777%	85.850%	82.562%	45.498%
8	89.777%	79.041%	99.446%	65.000%	16.620%	22.450%
16	88.148%	74.434%	99.783%	36.917%	5.657%	10.550%
20	87.821%	73.812%	N/A	N/A	3.523%	8.327%
24	87.593%	72.149%	N/A	N/A	2.411%	6.856%

Chapter 6

Conclusions

The prevailing trend in today's data center architectures is leveraging a large commodity parts to build high-performance clusters with tens of thousands of hosts, at a low cost. Unfortunately, in order to provide the necessary levels of communication bandwidth, the interconnects used in such clusters use expensive, non-commodity parts. Such solutions actually incur an almost exponential cost as the number of hosts increases, even when the designs allow links to be oversubscribed by a factor of 5 or more.

A novel data center network architecture has been recently proposed [8], which leverages commodity Ethernet switches, appropriately interconnected in a fat-tree topology, that uses a two-level routing function is able to provide full bisection bandwidth for large clusters, at a fraction of the cost of current best solutions. Touching mainly upon the implementation issues of this architecture, the focus of this thesis has been twofold:

- **Hardware implementation of the two-level lookup.** Chapter 4 presents two alternative designs for the two-level lookup. Both designs require constant time per lookup and can be easily implemented in hardware. One of these designs was implemented as a working prototype on the NetFPGA platform, with a slight modification of its router implementation. The implementation of the two-level lookup on the NetFPGA requires less than 100 lines of code and the experimental results have shown that this implementation of the two-level lookup does not impose any overhead over the traditional IP lookup mechanism.

- **Implementation of a network simulator.** Chapter 5 describes the implementation of a simulator for the fat-tree architecture. The simulator can be used to evaluate the performance of the architecture for an arbitrary size of the fat-tree and implements a variety of traffic patterns, including some of the worst-case communication patterns for the fat-tree architecture. Experimental results using the software simulator verified the theoretical expectations for the performance of the architecture.

The NetFPGA has proven to be a valuable tool for implementing and testing our proposed architecture. The implementation of the two-level on the NetFPGA platform has been a successful first step and it opens the way for the development of the more sophisticated routing techniques presented in the paper, namely the flow classification and flow scheduling techniques [8]. A characteristic of both of these techniques is that they require that the router keeps track of large flows. This can be done with sampling or hashing algorithms that require bounded memory [19]. Such a functionality can be straightforwardly added to the the existing NetFPGA router implementation.

Appendix A

BookSim code

```
/*

File: fattree.cpp
Author: Alexander Loukissas
Description: This file builds the topology for a 3-level fat tree out of 2k port
routers in the CVA simulator. Because of the design of the simulator and its
limitation to one-way channels, the router objects here have double the number
of ports.

*/

#include "booksim.hpp"
#include <vector>
#include <sstream>
#include <cassert>

#include "fattree.hpp"
#include "misc_utils.hpp"

//#define DEBUG_FAT_TREE
```

```
/* Constructor */
FatTree::FatTree( const Configuration &config ) : Network( config )
{
        _ComputeSize( config );
        _Alloc( );
        _BuildNet( config );
}

/* This function builds the topology: creates router objects and assigns input
and output channels to each object */
void FatTree::_BuildNet( const Configuration &config )
{
        ostringstream router_name;

        // there are 2*(k^2) routers in levels 0 & 1
        int per_stage = 2*powi( _k, 2 );

        // there are k^2 routers in level 2
        int center_stage = powi( _k, 2 );

        // node indexes the vector of router objects; routers are numbered
        // sequentially from level 0 to level 2 and from left to right
        int node = 0;

        int c;
        int level;

        /* Here we build the 2 lower stages of the fat tree (the pods) */
        for ( int stage = 0; stage < 2; stage++ ) {
        for ( int addr = 0; addr < per_stage; addr++ ) {
                router_name << "router_" << stage << "_" << addr;

#ifdef DEBUG_FAT_TREE
                cout << "creating " << router_name.str( ) << endl;
```

```
#endif
                _routers[node] = Router::NewRouter( config, this,
                        router_name.str( ), node, 2*_k, 2*_k );
                router_name.seekp( 0, ios::beg );
                _routers[node]->setIP();
                _routers[node]->populateTable( _k, stage );

#ifdef DEBUG_FAT_TREE
                cout << "connecting node " << node << " to:" << endl;
#endif


                /* iterate from 0 to _k-1 for the lower _k ports (level 0) */
                level = 0;
                for ( int port = 0; port < _k; ++port ) {
                /* input channels */
                        if ( stage == 0 ) {
                        // stage 0 routers have _k     inject channels in level 0
                                c = addr*_k + port;
                                _routers[node]->AddInputChannel(&_inject[c], &_inject_cred[c]);

#ifdef DEBUG_FAT_TREE
                        cout << " injection channel " << c << endl;
#endif
                        }
                        else {
                        // stage 1 routers have _k input channels in level 0 (from stage 0)
                                c = _InChannel( stage, level, addr, port );
                                _routers[node]->AddInputChannel( &_chan[c], &_chan_cred[c] );
#ifdef DEBUG_FAT_TREE
                        cout << " input channel " << c << endl;
#endif
                        }

                        /* output channels */
```

```
            if ( stage == 0 ) {
            // stage 0 routers have _k eject channels in level 0
                c = addr*_k + port;
                _routers[node]->AddOutputChannel( &_eject[c], &_eject_cred[c] );
#ifdef DEBUG_FAT_TREE
                cout << " ejection channel " << c << endl;
#endif
            }
            else {
            // stage 1 routers have _k output channels in level 0 (to stage 0)
                c = _OutChannel( stage, level, addr, port );
                _routers[node]->AddOutputChannel( &_chan[c], &_chan_cred[c] );
#ifdef DEBUG_FAT_TREE
                cout << " output channel " << c << endl;
#endif
            }
        }

        /* iterate from _k to (2*_k - 1) for the upper _k ports (level 1)*/
        level = 1;
        for ( int port = 0; port < _k; ++port ) {
        /* input channels */
            if ( stage == 0 ) {
            // stage 0 routers have _k     input    channels in level 1(from stage 1)

                c = _InChannel( stage, level, addr, port );
                _routers[node]->AddInputChannel( &_chan[c], &_chan_cred[c] );

#ifdef DEBUG_FAT_TREE
            cout << " input channel " << c << endl;
#endif
            }
            else {
            // stage 1 routers have _k input channels in level 1(from stage 2)
```

```
                    c = _InChannel( stage, level, addr, port );
                    _routers[node]->AddInputChannel( &_chan[c], &_chan_cred[c] );
#ifdef DEBUG_FAT_TREE
                cout << " input channel " << c << endl;
#endif
                }

            /* output channels */
                if ( stage == 0 ) {
                // stage 0 routers have _k output channels in level 1 (to stage 1)
                    c = _OutChannel( stage, level, addr, port );
                    _routers[node]->AddOutputChannel( &_chan[c], &_chan_cred[c] );
#ifdef DEBUG_FAT_TREE
                    cout << " output channel " << c << endl;
#endif
                }
                else {
                // stage 1 routers have _k output channels (to stage 2)
                    c = _OutChannel( stage, level, addr, port );
                    _routers[node]->AddOutputChannel( &_chan[c], &_chan_cred[c] );
#ifdef DEBUG_FAT_TREE
                    cout << " output channel " << c << endl;
#endif
                }
            }

            // node is an index to the routers[ ] array
            node++;
        }
        }

        /*  Here we build the third level of the fat tree (i.e. the core).
            Note that the core has half as many routers than the 2 lower levels. */
        int stage = 2;
```

```
    for ( int addr = 0; addr < center_stage; addr++ ) {
        router_name << "router_" << stage << "_" << addr;

#ifdef DEBUG_FAT_TREE
        cout << "creating " << router_name.str( ) << endl;
#endif

        _routers[node] = Router::NewRouter( config, this, router_name.str( ),
            node, 2*_k, 2*_k );
        router_name.seekp( 0, ios::beg );
        _routers[node]->populateTable( _k, stage );
        _routers[node]->setCore( true );

#ifdef DEBUG_FAT_TREE
        cout << "connecting node " << node << " to:" << endl;
#endif

        /* iterate from 0 to 2*_k-1 for the 2*_k ports in level 0 */
        level = 0;
        for ( int port = 0; port < 2*_k; ++port ) {
        /* input channels */
            // stage 2 routers have 2*_k input channels (from stage 1)
            c = _InChannel( stage, level, addr, port );
            _routers[node]->AddInputChannel( &_chan[c], &_chan_cred[c] );

#ifdef DEBUG_FAT_TREE
            cout << " input channel " << c << endl;
#endif

        /* output channels */
            // stage 1 routers have 2*_k output channels (to stage 1)
            c = _OutChannel( stage, level, addr, port );
            _routers[node]->AddOutputChannel( &_chan[c], &_chan_cred[c] );
#ifdef DEBUG_FAT_TREE
```

```
                cout << " output channel " << c << endl;
#endif
        }

        // node is an index to the routers[ ] array
        node++;
    }

}

/* This function computes the ingress channel index in a given port of a router
Inputs:
    stage: 0, 1 or 2, which corresponds to the level of the fat tree
    addr: horizontal position of the router in the given level
    level: for stage 0 or 1 routers, we conceptually discriminate between the
            lower and upper _k ports of the router
    port: the port of the router, ranging from 0 to _k - 1
Output:
    the index c in the array of 8*( _k^3 ) channels in the fat tree
*/
int FatTree::_InChannel( int stage, int level, int addr, int port ) const
{
    switch( stage ) {
        case 0: {
            assert( level == 1 );

            const int no_channels = 2*powi( _k, 3 );
            const int channel_offset = no_channels;
            const int cluster_offset = ( addr / _k ) * powi( _k, 2 );
            const int router_offset = addr % _k;

            return channel_offset + cluster_offset + router_offset + (port*_k);
        }
        case 1: {
```

```
        assert( level == 0 || level == 1 );

        const int no_channels = 2*powi( _k, 3 );
        // In lower level (0) there is 0x offset (from level 0)
        // In the upper level (1) there is 3x offset (from level 2)
        const int channel_offset = 3*( level * no_channels );
        if( level == 0 ) {
            const int cluster_offset = ( addr / _k ) * powi( _k, 2 );
            const int router_offset = addr % _k;
            return channel_offset + cluster_offset
                        + router_offset + (port*_k);
        }
        else {
            const int channels_per_cluster = 2*powi( _k, 2 );
            const int cluster_offset = (addr / (2*_k))*channels_per_cluster;
            const int router_offset = addr % (2*_k);
            const int port_offset = 2*_k * port;
            return channel_offset + cluster_offset
                        + router_offset + port_offset;
        }
    }
    case 2: {
        assert( level == 0 );

        const int no_channels = 2*powi( _k, 3 );
        const int channel_offset = 2 * no_channels;
        const int cluster_offset = addr;
        const int channels_per_cluster = powi( _k, 2 );
        const int port_offset = port * channels_per_cluster;

        return channel_offset + cluster_offset + port_offset;
    }
    default: return −1;
}
```

```
}

/* This function computes the egress channel index in a given port of a router
Inputs:
        stage: 0, 1 or 2, which corresponds to the level of the fat tree
        addr: horizontal position of the router in the given level
        level: for stage 0 or 1 routers, we conceptually discriminate between the
                        lower and upper _k ports of the router
        port: the port of the router, ranging from 0 to _k - 1
Output:
        the index c in the array of 8*( _k^3 ) channels in the fat tree
*/
int FatTree::_OutChannel( int stage, int level, int addr, int port ) const
{
        switch( stage ) {
                case 0: {
                        assert( level == 1);

                        const int channel_offset = 0;
                        const int cluster_offset = ( addr / _k ) * powi( _k, 2 );
                        const int router_offset = ( addr % _k ) * _k;

                        return channel_offset + cluster_offset + router_offset + port;
                }
                case 1: {
                        assert( level == 0 || level == 1 );

                        const int no_channels = 2*powi( _k, 3 );
                        // In lower level (0) there is 1*no_channels offset
                        // In upper level (1) there is 2*no_channels offset
                        const int channel_offset = ( stage + level ) * no_channels;
                        const int cluster_offset = ( addr / _k ) * powi( _k, 2 );
                        const int router_offset = ( addr % _k ) * _k;
                return channel_offset + cluster_offset + router_offset + port;
            }
            case 2: {
                assert( level == 0 );

                const int no_channels = 2*powi( _k, 3 );
                const int channel_offset = 3 * no_channels;
                const int channels_per_router = 2*_k;
                const int router_offset = addr * channels_per_router;

                return channel_offset + router_offset + port;
            }
            default: return −1;
        }
}


void FatTree::_ComputeSize( const Configuration &config )
{
  _k = config.GetInt( "k" );

  gK = _k;

  /* A fat-tree built out of 2*_k port routers can accomodate 2*(_k^3) hosts */
  _sources = 2*powi( _k, 3 );
  _dests    = 2*powi( _k, 3 );

  /* There are 2*(_k^2) routers in stages 0 & 1, and _k^2 routers in stage 2 */
  _size      = 5*powi( _k, 2 );

  /* There are 4*(_k^3) channels between stages 0 & 1 and stages 1 & 2 */
  _channels = 8*powi( _k, 3 );
}
```

```
int FatTree::GetK( ) const {return _k;}
```

Appendix B

NetFPGA programming script

```
#!/usr/bin/perl

use strict;
use NF2::TestLib;
use NF2::PacketLib;

my @interfaces = ("nf2c0", "nf2c1", "nf2c2", "nf2c3", "eth1", "eth2");
nftest_init(\@ARGV,\@interfaces,);
nftest_start(\@interfaces);

my $routerMAC0 = "00:ca:fe:00:00:01";
my $routerMAC1 = "00:ca:fe:00:00:02";
my $routerMAC2 = "00:ca:fe:00:00:03";
my $routerMAC3 = "00:ca:fe:00:00:04";

my $routerIP0 = "192.168.0.40";
my $routerIP1 = "192.168.1.40";
my $routerIP2 = "192.168.2.40";
my $routerIP3 = "192.168.3.40";

# clear LPM table
```

```
for (my $i = 0; $i < 32; $i++)
{
  nftest_invalidate_LPM_table_entry('nf2c0', $i);
}

# clear ARP table
for (my $i = 0; $i < 32; $i++)
{
  nftest_invalidate_ARP_table_entry('nf2c0', $i);
}

# Write the mac and IP addresses
nftest_add_dst_ip_filter_entry ('nf2c0', 0, $routerIP0);
nftest_add_dst_ip_filter_entry ('nf2c1', 1, $routerIP1);
nftest_add_dst_ip_filter_entry ('nf2c2', 2, $routerIP2);
nftest_add_dst_ip_filter_entry ('nf2c3', 3, $routerIP3);

nftest_set_router_MAC ('nf2c0', $routerMAC0);
nftest_set_router_MAC ('nf2c1', $routerMAC1);
nftest_set_router_MAC ('nf2c2', $routerMAC2);
nftest_set_router_MAC ('nf2c3', $routerMAC3);

print "So far so good\n";

# add an entry in the routing table:
my $index = 0;

# first terminating entry
my $subnetIP = "10.2.0.0";
my $subnetMask = "255.255.255.0";
my $nextHopIP = "10.2.0.1";
my $outPort = 0x1;

# second terminating entry
```

```
my $subnetIP2 = "10.2.1.0";
my $subnetMask2 = "255.255.255.0";
my $nextHopIP2 = "10.2.1.0";
my $outPort2 = 0x2;

# first non-terminating entry
my $subnetIP3 = "0.0.0.2";
my $subnetMask3 = "0.0.0.255";
my $nextHopIP3 = "10.4.1.1";
my $outPort3 = 0x3;

# second non-terminating entry
my $subnetIP4 = "0.0.0.3";
my $subnetMask4 = "0.0.0.255";
my $nextHopIP4 = "10.4.1.2";
my $outPort4 = 0x4;

my $nextHopMAC = "dd:55:dd:66:dd:77";

nftest_add_LPM_table_entry ('nf2c0',
                            1,
                            $subnetIP,
                            $subnetMask,
                            $nextHopIP,
                            $outPort);

nftest_add_LPM_table_entry ('nf2c0',
                            0,
                            $subnetIP2,
                            $subnetMask2,
                            $nextHopIP2,
                            $outPort2);

nftest_add_LPM_table_entry ('nf2c0',
                                0,
                                $subnetIP3,
                                $subnetMask3,
                                $nextHopIP3,
                                $outPort3);

nftest_add_LPM_table_entry ('nf2c0',
                                0,
                                $subnetIP4,
                                $subnetMask4,
                                $nextHopIP4,
                                $outPort4);

# add an entry in the ARP table
nftest_add_ARP_table_entry('nf2c0',
                             $index,
                             $nextHopIP,
                             $nextHopMAC);

# add an entry in the ARP table
nftest_add_ARP_table_entry('nf2c0',
                             1,
                             $nextHopIP2,
                             $nextHopMAC);

# add an entry in the ARP table
nftest_add_ARP_table_entry('nf2c0',
                                1,
                                $nextHopIP3,
                                $nextHopMAC);

# add an entry in the ARP table
nftest_add_ARP_table_entry('nf2c0',
                                1,
                                $nextHopIP4,
                                $nextHopMAC);

exit 0;
```

References

[1] First Steps Toward the Data Center of the Future - White Paper. http://www.dell.com/downloads/global/corporate/iar/20041101_meta.pdf.

[2] Myri-10G Product Overview. http://www.myri.com/Myri-10G/.

[3] Stanford Concurrent VLSI Architecture Group. http://cva.stanford.edu/.

[4] Sun Datacenter Switch 3456 Architecture White Paper. http://www.sun.com/.

[5] TOP500 Supercomputing Sites. http://www.top500.org/.

[6] InfiniBand®Architecture Specification Volume 1, Release 1.0, 2000.

[7] Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/, 2007.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM conference*, 2008.

[9] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, and J. Seizovic. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System

for Structured Data. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 205–218. USENIX Association, 2006.

[11] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov. Characteristics of Workloads used in High Performance and Technical Computing. In *ICS '07: Proceedings of the 21st International Conference on Supercomputing*, pages 73–82. ACM, 2007.

[12] J. Chu and V. Kashyap. Transmission of IP over InfiniBand (IPoIB). RFC 4391, Internet Engineering Task Force, 2006.

[13] C. Clos. A study of Non-Blocking Switching Networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[14] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI' 04: Proceedings of the 6th Symposium on Opearting Systems Design & Implementation*, pages 137–149. USENIX Association, 2004.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP '07: Proceedings of 21st Symposium on Operating Systems Principles*, pages 205–220. ACM, 2007.

[17] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. *ACM SIGCOMM Computer Communications Review*, 27(4):3–14, 1997.

[18] W. Eatherton, G. Varghese, and Z. Dittia. Tree Bitmap: Hardware/Software IP lookups with Incremental Updates. *ACM SIGCOMM Computer Communications Review*, 34(2):97–122, 2004.

[19] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing tn the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 2003.

[20] M. Feldman. Cisco Delivers Integrated Ethernet/InfiniBand Framework. `http://www.hpcwire.com/hpc/707228.html`.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP '03: Proceedings of the 19th Symposium on Operating Systems Principles*, pages 29–43. ACM, 2003.

[22] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, Internet Engineering Task Force, 2000.

[23] V. Kashyap. Dynamic Host Configuration Protocol (DHCP) over InfiniBand. RFC 4390, Internet Engineering Task Force, 2006.

[24] V. Kashyap. IP over InfiniBand: Connected Mode. RFC 4755, Internet Engineering Task Force, 2006.

[25] V. Kashyap. IP over InfiniBand (IPoIB) Architecture. RFC 4392, Internet Engineering Task Force, 2006.

[26] T. T. Kwan, B. K. Totty, and D. A. Reed. Communication and Computation Performance of the CM-5. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 192–201, 1993.

[27] C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, D. Hillis, B. Kuszmaul, M. Pierre, D. Wells, et al. The Network Architecture of the Connection Machine CM-5 (Extended Abstract). In *SPAA '92: Proceedings of the 4th Symposium on Parallel Algorithms and Architectures*. ACM, 1992.

[28] C. E. Leiserson. Fat-trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, 1985.

[29] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–An Open Platform for Gigabit-rate Network Switching and Routing. In *MSE '07: Proceedings of the IEEE International Conference on Microelectronic Systems Education*, pages 160–161, 2007.

[30] J. Moy. OSPF Version 2. RFC 2328, Internet Engineering Task Force, 1998.

[31] H. Noda, K. Inoue, M. Kuroiwa, F. Igaue, K. Yamamoto, H. Mattausch, T. Koide, A. Amo, A. Hachisuka, S. Soeda, et al. A Cost-Efficient High-Performance Dynamic TCAM with Pipelined Hierarchical Searching and Shift Redundancy Architecture. *IEEE Journal of Solid-State Circuits*, 40(1):245–253, 2005.

[32] K. Pagiamtzis and A. Sheikholeslami. A Low-Power Content-Addressable Memory (CAM) Using Pipelined Hierarchical Search Scheme. *IEEE Journal of Solid-State Circuits*, 39(9):1512–1519, 2004.

[33] K. Pagiamtzis and A. Sheikholeslami. Using Cache to Reduce Power in Content-Addressable Memories (CAMs). In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 369–372, 2005.

[34] K. Pagiamtzis and A. Sheikholeslami. Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, 2006.

[35] V. Ravikumar, R. Mahapatra, and L. N. Bhuyan. EaseCAM: An Energy and Storage Efficient TCAM-based Router Architecture for IP Lookup. *IEEE Transactions on Computers*, 54(5):521–533, 2005.

[36] Y. Rekhter and T. Li. An Architecture for IP Address Allocation with CIDR. RFC 1518, Internet Engineering Task Force, 1993.

[37] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous. Survey and Taxonomy of IP Address Lookup Algorithms. *IEEE Network*, 15(2):8–23, 2001.

[38] D. Shah and P. Gupta. Fast Updating Algorithms for TCAMs. *IEEE Micro*, 21(1):36–47, 2001.

[39] V. Srinivasan and G. Varghese. Fast Address Lookups using Controlled Prefix Expansion. *ACM Transactions on Computer Systems (TOCS)*, 17(1):1–40, 1999.

[40] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991, Internet Engineering Task Force, 2000.

[41] L. G. Valiant and G. J. Brebner. Universal Schemes For Parallel Communication. In *STOC '81: Proceedings of the 13th ACM Symposium on Theory of Computing*, pages 263–277, 1981.