Physically Dense Server Architectures

by

Anthony Thomas Gutierrez

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2015

To my mother.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my advisor Trevor Mudge, without whom this thesis would not be possible. His laid back style allowed me to explore research topics for myself, however he was always there to provide feedback, guidance, equipment, and whatever else I needed. Being a student of Trev's I was able to travel to conferences, collaborate with industry leaders on interesting projects, work with well-respected academic researchers at Michigan as well as other top universities, and enjoy many lunches at Casey's. Ronald Dreslinski was also a big help to me going back to the first day I joined Trev's research group, when he was still a fellow student. He became like a co-advisor for all of us in the (Trev + Ron) TRON group. I also had the pleasure to work with Thomas Wenisch on several projects during my graduate career. Tom's encyclopedic knowledge and ability to explain things simply was always useful. In addition I would like to thank the remaining members of my committee: Jason Mars and David Blaauw, who provided me with valuable feedback on this dissertation.

I would be remiss if I didn't acknowledge my fellow TRON group lab members over the years: Nilmini Abeyratne, Geoff Blake, Yajing Chen, Mike Cieslak, Cao Gao, Johann Hauswald, Yiping Kang, Tom Manville, Mark Woh, and Mr. Qi Zheng. We had some good times together.

There were many friends who helped keep me sane throughout my graduate career, and special thanks goes out to them. Joe Pusdesris and I formed what will be a lifelong friendship as soon as he joined the TRON group as an undergrad—earning him the moniker ”UG” Joe. Whether it was eating sloppy chili dogs at C-Reds, or checking out some local concerts, UG and I always had a good time. Lots of good food, beer, and conversation were had with Matt Burgess, Jason Clemons, Joe Greathouse, Rick Hollander, Dave Meisner, and Andrea Pellegrini. Korey Sewell took me under his wing when I first joined Trev’s group and we quickly became friends, playing pickup basketball a few times a week, and tailgating with the rest of the game day crew on football Saturdays. And not to forget the rest of the game day crew, I must thank Ced Armand, Kevin ”Sleep” Carter, Lou Gilling, and Chuck Sutton. We had a lot of memorable times over the years, and I’m looking forward to a lot more.

To my girlfriend Alyssa Doffing, thanks for sticking by me throughout this entire process. It’s finally over.

Finally I thank my family for all of their love and support. First I must thank my mother Cheryl Gutierrez, whose love and sacrifice inspired me to become the person I am today. My grandmother Theresa Coutinho is one of the most kind and hard working people I have ever met, without her support I would not have made it this far. My brother Carlos Gutierrez and my sister Chelsea Gutierrez were always there for me no matter how hard times got. I love you all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ARP address resolution protocol

BGA ball grid array

CQ completion queue

DHT distributed hash table

DMA direct memory access

HCA host channel adapter

HMC Hybrid Memory Cube

IP internet protocol

KVD key-value distributor

LRG least-recently-granted

NIC network interface controller

OoO out-of-order

PHY physical

QoS quality of service

QP queue pair

RDMA remote direct memory access

RPS requests per second

RTT round-trip time

Rx receive

SLA service-level agreement

SoC	system on a chip
TCP	transport control protocol
TSV	through-silicon via
Tx	transmit
VIP	virtual IP
WQE	work queue element

ABSTRACT

Physically Dense Server Architectures

by

Anthony Thomas Gutierrez

Chair: Trevor N. Mudge

Distributed, in-memory key-value stores have emerged as one of today's most important data center workloads. Being critical for the scalability of modern web services, vast resources are dedicated solely to key-value stores in order to ensure that quality of service guarantees are met. These resources include: many server racks to store terabytes—possibly petabytes—of key-value data, the power necessary to run all of the machines, networking equipment and bandwidth, and the data center warehouses used to house the racks.

There is, however, a mismatch between the key-value store software and the commodity servers on which it is run, leading to inefficient use of resources. The primary cause of this inefficiency is the overhead incurred from processing individual network packets, which typically carry small payloads of less than a few kilobytes, and require minimal compute resources. Thus, one of the key challenges as we enter the exascale era is how to best adjust to the paradigm shift from compute-centric data centers, to storage-centric data centers.

This dissertation presents a hardware/software solution that addresses the inefficiency issues present in the modern data centers on which key-value stores are currently deployed. First, it proposes two physical server designs, both of which use 3D-stacking technology and low-power CPUs to improve density and efficiency. The first 3D architecture—Mercury—consists of stacks of low-power CPUs with 3D-stacked DRAM, as well as NICs. The second architecture—Iridium—replaces DRAM with 3D NAND Flash to improve density.

The second portion of this dissertation proposes and enhanced version of the Mercury server design—called KeyVault—that incorporates integrated, zero-copy network interfaces along with an integrated switching fabric. In order to fully utilize the integrated networking hardware, as well as reduce the response time of requests, a custom networking protocol is proposed. Unlike prior works on accelerating key-value stores—e.g., by completely bypassing the CPU and OS when processing requests—this work only bypasses the CPU and OS when placing network payloads into a process' memory. The insight behind this is that because most of the overhead comes from processing packets in the OS kernel—and not the request processing itself—direct placement of packet's payload is sufficient to provide higher throughput and lower latency than prior approaches. The need for complex hardware or software is also eliminated.

CHAPTER I

Introduction

We have now entered into an era with unprecedented amounts of data stored in the cloud. A recent research report from Nasuni estimates that there is now over one exabyte of data stored in the cloud [69], while Facebook reports an incoming data rate of approximately 600TB per day [83]. Amazon alone reports that their S3 cloud storage service contains 1 trillion objects—that is 142 objects for each person on Earth [6]. It is interesting to note that, while there is an estimated one exabyte of data in the cloud, the average object size is quite small. Figure 1.1 shows a file size distribution of files stored by a typical user, as reported by Nasuni [1]. Over 77% of files are one megabyte or less, and over 67% are 100 kilobytes or less. What this tells us is that not only do cloud services need to store and manage vast amounts of data, but they also must serve many requests for small, disparate pieces of data. Given the large cost of building, maintaining, and running a modern data center warehouse, the key challenge becomes how to best adjust to the paradigm shift to data-centric computing.

To ensure quality of service (QoS) guarantees are met, cloud services often cache objects in memory to prevent slow disk lookups for latency-critical requests. The greater performance of DRAM when compared to disks comes with a cost—higher prices and far less memory density. The irony of this fact is two-fold: first, because

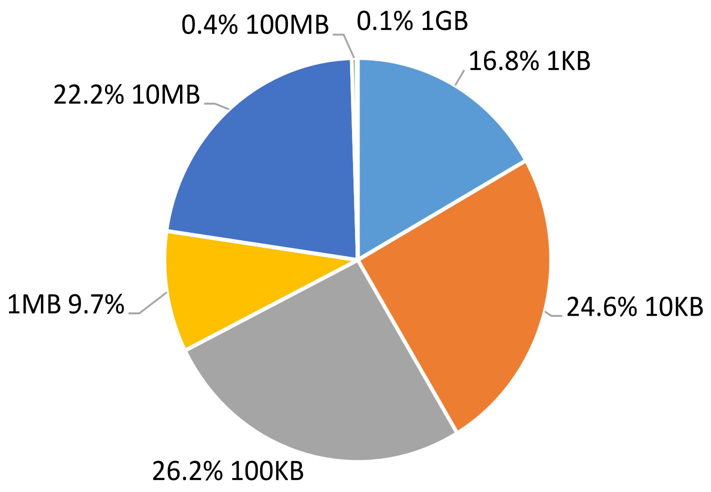


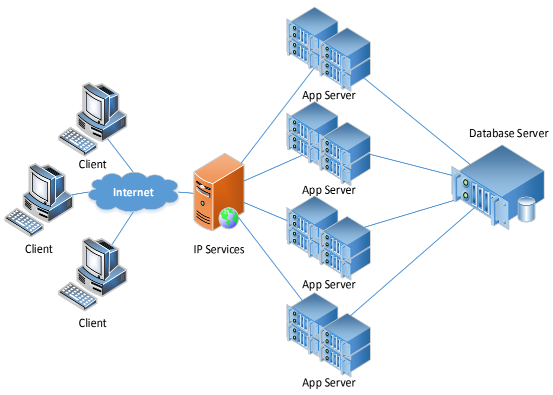
Figure 1.1: File size distribution of files stored by a typical user.

of the vast data sets on which cloud services operate, in-memory caching forces them to be highly distributed. This removes much of the benefit of using DRAM because requests must now traverse the network. And second, data centers are typically constructed using commodity off-the-shelf-hardware, thus adding more DRAM is not a viable solution for scalability due to its high cost.

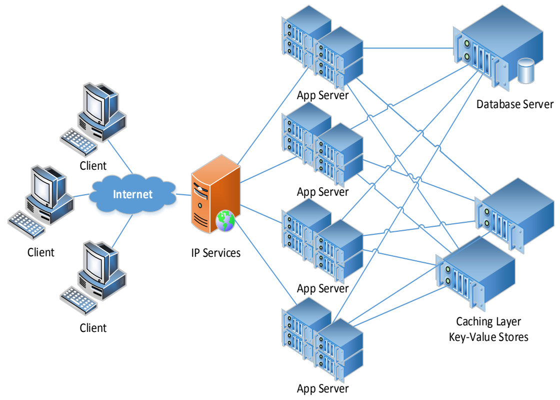
1.1 Scaling in the Cloud

Since the emergence of the Web 2.0 era, scaling internet services to meet the requirements of dynamic content generation has been a challenge—engineers quickly discovered that creating content for each visitor to a web site generated a high load on the back-end databases. While it is easy to scale the number of servers generating HTML and responding to client requests, it is harder to scale the data stores. This two-tier infrastructure—as shown in figure 1.2a—becomes increasingly difficult to scale and requires many redundant systems to prevent a single point of failure. In order to meet the performance and reliability requirements of handling such a massive volume of data, highly distributed scale-out architectures are required.

Internet services operate on data sets that are more massive than ever before,



(a) DB handles all reads and writes.



(b) DB handles only reads and writes that miss in the cache.

Figure 1.2: Two different models for web servers.

and are responsible for servicing many real-time requests to small, disparate pieces of data. As mentioned, servicing these requests from the back-end databases does not scale well; fortunately the size and popularity of requested objects exhibit power-law distributions [3, 17], which makes them suitable for in-memory caching. Figure 1.2b shows a web server with the addition of in-memory caching. As we enter the peta-scale era, key-value stores will become one of the foremost scale-out workloads used to improve the scalability of internet services. As such, vast data center resources will be dedicated solely to key-value stores in order to ensure that QoS guarantees are met.

Memcached is one example of a distributed, in-memory key-value store caching system. Memcached is used as the primary piece of scaling infrastructure for many of today's most widely-used web services, such as Facebook, Twitter, and YouTube. Due to its wide-spread use, and the high cost of data center real estate, it is important that Memcached be run as efficiently as possible. Today, however, Memcached is deployed on commodity hardware consisting of aggressive out-of-order cores, whose performance and efficiency are measured with respect to how well they are able to run CPU benchmarks, such as SPEC [31].

Key-value stores are currently deployed on commodity off-the-shelf hardware, which is not ideal due to numerous inefficiencies caused by a mismatch with the application domain. The highly distributed nature of key-value stores requires frequent remote memory accesses, which can be several orders of magnitude slower than local memory when nodes are connected using commodity networking technology. Several prior works have shown that key-value store applications do not fully utilize high-end out-of-order (OoO) CPUs—exhibiting poor cache hit rates, and high branch mis-prediction rates—and do not come close to saturating available memory or network bandwidth [2, 22, 23, 56]. The primary cause of this inefficiency is the overhead associated with processing individual network packets, which typically carry small payloads on the order of kilobytes. The large code footprints associated with the networking stack and operating system code cause a bottleneck in the instruction fetch portion of the CPU; and complex, off-chip NICs cause slow data transfers.

As the bandwidth of network interfaces approaches 1Tb/s [87]—matching the available bandwidth of the CPU's memory controller—the problem of network packet processing will be exacerbated. The overhead of relying on the CPU and OS to process packets, and get data off chip, will be the limiting factor to fully utilizing available network bandwidth. To ensure QoS requirements are met, on-chip networking hardware with DMA capabilities—along with lightweight networking protocols—will be crucial.

1.2 Data Center Trends

Realizing that aggressive out-of-order cores are not an efficient choice for many classes of server applications, several studies have advocated the use of low-power embedded CPUs in data centers: [2, 42, 57, 76]. There are challenges with this approach as well—embedded cores are unable to provide the throughput and latency guarantees that are required to supply responsive dynamic content. More recently, Lim et al. [54] have shown that mobile cores alone are not enough to improve the efficiency of distributed key-value store caching systems. Instead, they create a custom Memcached accelerator, and rely on recent technology scaling trends to closely integrate their accelerator into a system on a chip (SoC) design they call Thin Servers with Smart Pipes (TSSP). These approaches may improve overall energy and performance, however they do not address density. Dense servers, on the other hand, have been shown to provide greater efficiency for CPU-bound electronic commerce workloads in the past [21], however for workloads that required large amounts of memory, traditional servers were found to be a better solution.

As the volume of data in the cloud increases, there is a growing trend towards distributing large data sets across many servers and pushing the compute—e.g., database queries, key-value GETs, etc.—to the data. The highly distributed nature of these applications requires that most data accesses must be done over the data center's network, which is a source of significant delay [79].

Data centers are provisioned to meet peak demand, ensure low variability for response times, and provide fault tolerance. Current data centers employ commodity hardware because of its cost-effective performance. This leads to data center designs that push the limits of power supplies, warehouse space, and cooling, without fully utilizing the available compute, storage, or communication resources; data centers typically operate between 10-50% utilization on average [8].

1.2.1 Opportunities for Improved Efficiency

Researchers have noted the aforementioned issues and there is now a move towards integration, improved form factor, and higher density. Several trends have emerged:

1. **Modular data centers.** The idea of modular and portable data centers has been around for some time. Sun's Project Blackbox [84] aimed to put data centers in portable shipping containers; the idea being that a data center could

be housed in a portable unit at a fraction of the cost of a warehouse, and without a long term commitment to the location. Google has even patented water-based data centers [16]. The idea of housing data centers in portable spaces naturally leads to more compact rack-scale servers. HP's Moonshot [34] and current offerings from SeaMicro [19] are examples of such designs. Both of which recognize the need for dense form factor and integrated networking technology.

2. **Storage-focused data centers.** Realizing that the growth of data in the cloud is far outpacing the ability of disk-based storage to scale, works such as *RAMCloud* [72] propose storage entirely in DRAM. The idea of using DRAM for storing large data sets, has led to works on alternative, storage-focused data center architectures. Alternative memory technologies such as 3D-stacked DRAM and non-volatile memory have been proposed for use in data centers to improve energy efficiency over commodity components [43, 45, 77]. *PicoServer* [42] was one of the first works to propose using 3D-stacked memory to improve energy efficiency and density. Even more exotic 3D-stacked architectures, such as *nanostores* [15], put 3D-stacked compute on top of non-volatile memory layers. Work by Lim et al. even suggests using *memory blades*: disaggregated memory servers with minimal compute capabilities [55].

3. **Integrated Networking.** As data becomes more distributed and the amount of compute shrinks, the network is becoming the bottleneck for data centers applications. Deep network stacks, queueing, and off-chip networking devices become the limiting factor when trying to meet latency and throughput guarantees. While commodity NICs provide a cost-effective solution, and protocols such as TCP/IP and UDP are well supported, there are many unexplored opportunities to improve network performance. Binkert et al. demonstrate how

simple, integrated NIC designs can improve the performance of TCP/IP [11], while RDMA has become a popular solution for extremely high-bandwidth, low latency communication for in-memory data center applications [38, 39, 66].

4. **The use of "wimpy" cores in servers.** Andersen et al. noted the opportunity to improve throughput and saving energy with *FAWN* [2]. FAWN uses "wimpy" cores and FLASH memory to reduce the amount of energy per operation density. Felter et al. prototyped *Super Dense Servers* [21], which combine low-power cores and a tight form factor as a way to provide fine-grain power management in response to variations in load. Still others propose using sophisticated schemes to manage power. In *PowerNap* [60] tasks are delayed to create idle periods sufficiently long enough to enable cores to be put into deep sleep modes. Recent work by Lo et al. [58] proposes a power management approach that utilizes machine learning to manage DVFS in order to improve the power efficiency of data centers. The common theme in these works is that OoO CPUs are over provisioned for modern data-centric data center workloads.

Several common themes emerge from the work on data center efficiency: 1) there is a need for higher levels of integration; 2) current power management techniques are inadequate; 3) commodity, off-the-shelf hardware—while cost-effective—is not viable going forward; 4) we need lower overhead networking; and finally 5) storage density needs to improve. While these problems have been addressed in isolation, and in the context of current data center designs, we believe a holistic approach is necessary.

1.3 Contributions of this Dissertation

In this dissertation the notion of an efficient, integrated system is taken one step further to include density as a primary design constraint. First, it explores the design of two integrated, 3D-stacked architectures called Mercury and Iridium[1]. With Mercury, low-power cores similar to ARM Cortex-A7s are tightly coupled with NICs and DRAM, while maintaining high bandwidth and low latency. Recently, Facebook introduced McDipper [25], a Flash-based Memcached server using the observation that some Memcached servers require higher density with similar latency targets, but are accessed at much lower rates. To address these types of Memcached servers, a server designed called Iridium is introduced, a Flash based version of Mercury that further increases density at the expense of throughput while still meeting latency requirements. These two architectures allow density to be significantly increased, resulting in more effective use of costly data center space.

Next, simple, integrated networking hardware is added that allows for a low-overhead—yet general purpose and powerful—networking protocol. This server design, KeyVault, builds upon the Mercury server design extending it to include simple, zero-copy NICs along with an integrated switch fabric and a lightweight communication protocol. In summary, this dissertation makes the following contributions:

- Given data center costs, this work contends that server density should be considered a first-class design constraint.

- It Mercury, an integrated, 3D-stacked DRAM server architecture which has the potential to improve density by 2.9×, making more efficient use of costly data center space.

- By closely coupling DRAM and NICs with low-power Cortex-A7-like cores, it show that it is possible to improve power efficiency by 4.9×.

- By increasing density, and closely coupling DRAM, NICs, and CPUs, it is possible to increase requests per second (RPS) by 10× and TPS/GB by 3.5×.

[1]Mercury, the Roman god, is extremely fast, while the element of the same name is very dense. Iridium, while not a god, is more dense than mercury.

- It proposes Iridium for McDipper [25] style Memcached servers, which replaces the DRAM with NAND Flash to improve density by 14×, TPS by 5.2×, and power efficiency by 2.4×, while still maintaining latency requirements for a bulk of requests. This comes at the expense of 2.8× less TPS/GB due to the much higher density.

- It makes the observation that, in order to achieve maximum performance, it is not necessary to bypass the CPU and OS completely when processing requests. Performing zero-copy packet transmission is enough to outperform solutions that completely bypass the CPU.

- Design an integrated NIC that allows for true zero-copy packet transmission.

- Specify a networking protocol that makes full use of the integrated NICs, thereby reducing the round-trip time (RTT) of requests, and improving the max sustainable RPS.

- Design an integrated switching fabric that allows the network links to be shared amongst several stacks, thereby improving the network link utilization.

- Provide a detailed design space exploration of various 1.5U KeyVault server designs, contrasted with prior works.

The rest of this dissertation is organized as follows: in chapter II background information related to computer networking—particularly in the context of high-performance data centers—including both traditional TCP/IP networks and modern RDMA networks. Background information pertaining to near-data compute technology is also discussed. In chapter III describes the Mercury and Iridium server designs in detail. It provides an evaluation of each server design while running Memcached, and performs a detailed design space exploration. The work presented in chapter III was presented at ASPLOS 2014 [29]. Chapter IV describes how the design of the Mercury server was modified to include an integrated switching fabric, integrated network interfaces with zero-copy capabilities, along with a low-overhead networking protocol. Finally, chapter V provides concluding remarks and proposes future research directions.

CHAPTER II

Background

This chapter outlines some of the fundamental knowledge regarding networking and near-data compute technology in order to understand the work presented in this thesis. First, traditional TCP/IP networks are discussed, followed by a discussion of current high-performance networks that use remote direct memory access (RDMA). Finally an overview of processing-in-memory technology as well as processing-near-memory technology is given.

2.1 TCP/IP Networking

Figure 2.1 shows the network stack for the TCP/IP protocol suite. This model has essentially four software layers on top of the physical layer, and is the most widely-used protocol suite in computer networking.

2.1.1 Application Layer

The ultimate goal of a computer network is to be able to exchange data between two communicating processes; this is shown as the top layer in figure 2.1. Data transmission and retrieval begins at the *application layer*. In this layer, processes package the data that is to be transmitted across the network into a packet with a previously agreed upon format. E.g., the packet may contain headers with information

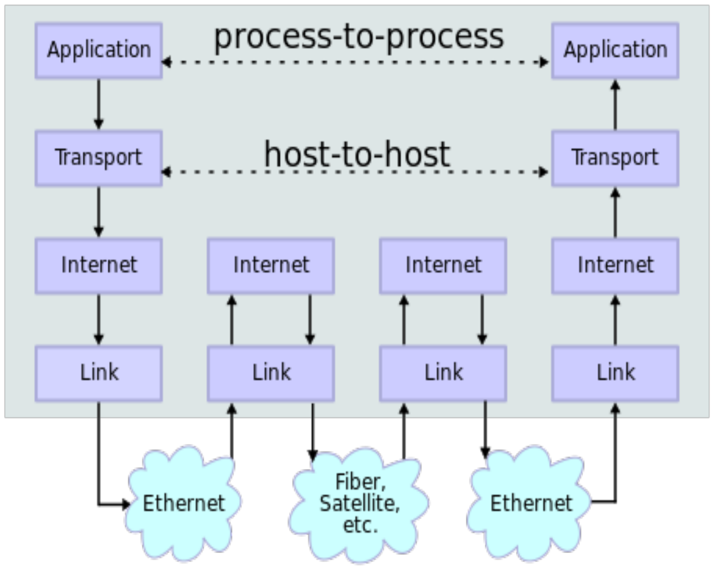


Figure 2.1: Network layers for the TCP/IP model.

on the data type, length, format, etc. In this dissertation, the application of interest is Memcached [62], and its application layer packet contains headers that describe the type of request—GET, SET, DELETE, etc.—the size of the key and value, and error checking codes [63].

Packets are sent to the next through the *sockets* API, which is a buffering layer that provides support for several transmission protocols via a standard interface. The sockets layer is responsible for copying data from the application's memory space to kernel space and vice versa. The sockets layer can be blocking or non-blocking—meaning the application may wait on a send or receive operation until space is freed, or data is available respectively. If the call is non-blocking then it will continue on and attempt to send or receive data at a later time. The bottleneck in this processes is the copy of packet data from kernel space to user space, because these reads typically cause many cache misses. Techniques that place packet data directly into the cache [35] may not provide as much benefit as they could do to cache interference with application data.

2.1.2 Transport Layer

The next layer in the networking stack is the *transport layer*, and this layer typically uses the transport control protocol (TCP) layer [13]. The transport layer is responsible for establishing a channel between two host machines. This is done by establishing a virtual connection between two host machines and assigning each a unique port number. The application layer sends a stream of data to the TCP layer and it is responsible for diving the data stream up into packets and sending those packets to the next level in the stack.

Because—as we will see in section §2.1.3—the *internet layer* does not guarantee that packets are delivered in order, or even delivered at all, TCP is responsible for assuring retransmission and reordering of packets when necessary. TCP also provides some error checking via checksums, flow control, receive acknowledgement, and is able to detect packet duplication. Because packets are sent to a best-effort network over lossy links, it is a very heavyweight software layer.

Because TCP is a reliable protocol, it must buffer packets in kernel space memory to ensure that no packets are dropped at this layer. It uses a sliding window scheme to ensure that there are never more outstanding bytes awaiting transmission than there is buffer space at a receiver.

2.1.3 Internet Layer

The internet layer typically uses the internet protocol (IP) [13]. The IP is a best-effort datagram protocol; it does not guarantee in-order delivery, or even that packets are delivered at all. The IP also does not prevent packet duplication. The IP packet contains the IP address of the source and destination, as well as information indicating which transport layer protocol is being used. This information is used to route the packets to their destinations. If the packet is larger than the maximum allowable size, the IP will split the packets up into separate fragments.

2.1.4 Link Layer

The lowest layer in the TCP/IP suite is the *link layer*. The link layer consists of several protocols and is primarily responsible for address resolution via the address resolution protocol (ARP). At the link layer packets are typically encapsulated in Ethernet frames. These frames contain the Ethernet addresses—which will be used by the ARP to convert IP addresses to hardware addresses—for both the source and destination devices. The link layer interacts with the network interface controller (NIC) via its device driver.

The NIC driver is responsible for sending packets to the device and signaling that packets are available to be transmitted, as well as transferring packets that are written into memory by the device to the application. The driver sets up *descriptor rings* in memory where data may be exchanged between the hardware and the software. The descriptor rings are essentially circular buffers where packets may be read from, and written to.

When transmitting a packet, the driver places the packet address, or the packet itself, into an entry in the ring descriptor and updates the ring's next pointer. It then signals to the device that a packet is read to send. The NICs direct memory access (DMA) engines read the packet data from memory and send it over the physical interface.

Once a packet is received, the device notifies the CPU that a packet is ready to be read; this is done either by using an interrupt or by polling. The driver will then read the packet from memory and it will be sent to the network stack for processing, eventually being copied from kernel space memory to the application's memory.

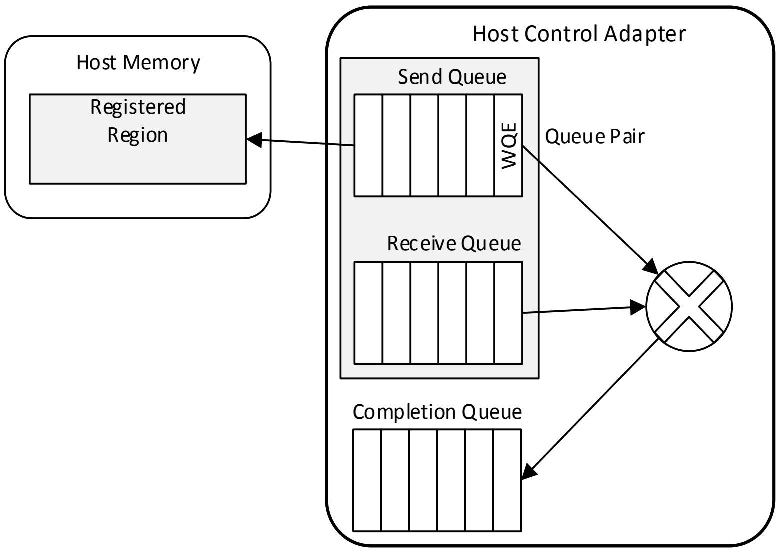


Figure 2.2: The work queues inside an HCA adapter.

2.2 RDMA over InfiniBand

InfiniBand is a high-performance network communications link, primarily used in data centers and storage servers. It is essentially a high-performance replacement for Ethernet. InfiniBand uses a switched fabric topology, and each processor contains an host channel adapter (HCA). The key advantages of InfiniBand over Ethernet are: its high-speed links, scalable fabric, and RDMA capabilities.

RDMA over InfiniBand uses a message passing protocol to allow for zero-copy networking—allowing the HCA to directly transfer data to or from the application's memory. reads and writes are performed directly by the HCAs using message passing *verbs*. The RDMA verbs may either be one-sided—the host CPU is completely unaware of the data transfer being performed by the HCA—or they may be two-sided—the host CPU must explicitly receive the data from the HCA. The one-sided verbs have lower overhead because the CPU is involved, however they may cause synchronization issues if the host CPU is reading or writing the data as the HCA is modifying it.

The HCA maintains several hardware queues for sending and receiving packet data. Figure 2.2 shows these queues. There are two work queues: a send queue and a receive queue. Together these queues are known as a queue pair (QP). The HCA driver prepares a work queue element (WQE) in a dedicated region of the host's memory, then it notifies the HCA that a verb is ready to be processed. The WQEs contain all the information needed to process the verbs. Once the HCA has processed the verb it stores a completion event into the completion queue (CQ) via a DMA write.

At the transport layer, RDMA can be connected or connectionless. In the case of a connected transport, two QPs must be directly connected via a virtual connection, and communicate only with each other. At the link layer, RDMA packets are transmitted over a lossless fabric, meaning packets are never dropped due to buffer overflows.

2.3 Near-Data Compute

This section describes the prior efforts related to integrating compute with memory, as well as the current state-of-the-art technology used to put compute near memory: 3D-stacked memory.

2.3.1 Processing in Memory

As far back as the early 90s researchers recognized the potential of putting compute very near to memory, in order to overcome the limitations of the *memory wall*, primarily providing enough bandwidth to keep compute units fed for data parallel workloads. These works proposed mixing compute and memory directly [48, 49, 50]. However, the technology of the time made processing in memory infeasible due to its high cost and low memory density.

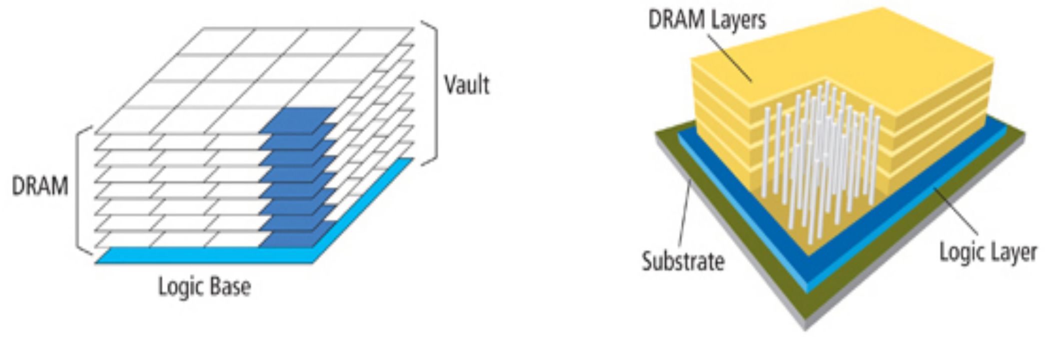


Figure 2.3: A high-level overview of Micron's Hybrid Memory Cube technology.

2.3.2 3D-Stacked Memory Technology

Recent technology have advances have made it possible to—through the use of through-silicon via (TSV) technology—integrated memory and logic vertically. Because memory and logic are not directly mixed, the cost is not as high as putting compute in memory; density is also greatly improved. 3D-stacking does, however, allow for very high bandwidths, and low latency, while also improving energy efficiency.

Micron's Hybrid Memory Cube (HMC) [73] and Tezzaron's Octopus [82] are two such efforts that seems very promising. Figure 2.3 shows a high-level overview of Micron's HMC technology. In this figure we can see that several layers of DRAM are connected to a logic layer using TSVs. This technology, e.g., can provide over 100GB/s of bandwidth at a fraction of the power of conventional DRAM [73].

Given the promise of 3D-stacked memory, there has been a renewed interest in processing-near-memory architectures—particularly for big data workloads which, given their small compute to data ratio, seem well-suited for 3D-stacked architectures. Many recent works have looked at using 3D-stacked memory in the context of big data workloads with promising results: [15, 29, 42, 43, 45, 75].

Given the promise of utilizing alternative memory and networking technologies, this dissertation argues that we will need to move away from commodity off-the-shelf hardware to build our data centers. The benefits will be improved performance, better space utilization, and better energy efficiency.

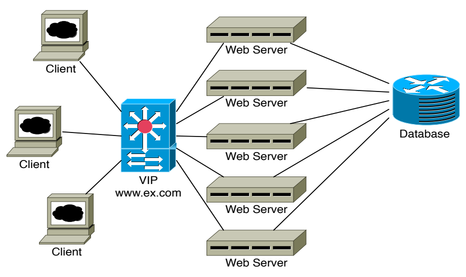
CHAPTER III

Integrated 3D-Stacked Server Architectures

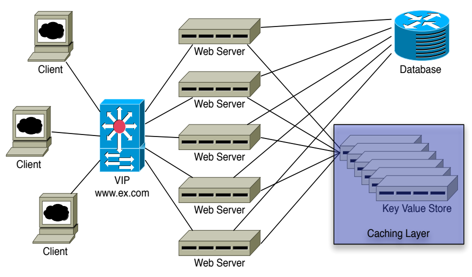
3.1 Background

3.1.1 Cloud Computing Design

Figure 3.1a shows the design of a standard web server setup. A load balancer typically has one or more virtual IP (VIP) addresses configured. Each domain is routed by a DNS to a particular load balancer, where requests are then forwarded to a free server. After the load balancer a fleet of front-end servers service the web request. If needed, these web servers will contact a back-end data store to retrieve custom content that pertains to the user, which means that all servers could connect to the back-end data store.



(a) The database handles all reads and writes.



(b) The database only handles writes and reads that miss in the cache.

Figure 3.1: Configurations with 2 and 3 layers behind a vip to service web requests.

In Figure 3.1b a caching layer is added, which can service any read request that hits in the cache. A request will first be forwarded to the caching layer on any read. If the requested data is present, the caching layer returns its copy to the client. If the data is not present, a query will be sent to the back-end data store. After the data returns, the server handling the request will issue a write to the caching layer along with the data. Future requests for that data can then be serviced by the caching layer. All write requests are sent directly to the back-end data store. Updating values in the caching layer depends on how it is configured. The two most common cases are that writes are duplicated to the caching layer or a time-to-live is placed on data in the caching layer.

3.1.2 Scaling in the Cloud

In general, the traffic to a website varies by the time of day and time of year. Data published by Netflix [71] demonstrates how traffic to their site varies over a three day period. As their data shows, traffic peaks during midday, and is at its lowest point around midnight. They also quantify the corresponding number of front-end servers needed to maintain equal load throughout the day, which tracks closely with the traffic. While this is easy to do for front-end servers, because they maintain little state, back-end data stores are not easily scaled up and down. Netflix overcomes this problem by the extensive use of caching, which is a cheaper solution than scaling back-end data stores.

While turning on and off servers helps save power, it does not help reduce space because the servers must still be physically present in order to meet peak demand. To cope with the physical demands of scaling, new data centers must be built when a given data center is either over its power budget, or out of space. A recent report states Google is spending 390 million USD to expand their Belgium data center [74]. Facebook also has plans to build a new 1.5 billion USD center in Iowa[64]. Because of this high cost, it is critical to avoid scaling by means of simply building new data centers or increasing the size of existing ones. This paper focuses on increasing physical density within a fixed power budget in order to reduce the data center footprint of key-value stores.

3.1.3 Memcached

In this paper we use Memcached 1.4 as our key-value store. We choose Memcached as our key-value store because of its widespread use in cloud computing. Memcached does not provide data persistence and servers do not need to communicate with each other, because of this it achieves linear scaling with respect to nodes. To date, the largest Memcached cluster with published data was Facebook's, which contained over 800 servers and had 28TB of DRAM in 2008 [89].

The ubiquity of Memcached stems from the fact that it is easy to use, because of its simple set of verbs. Only three details about Memcached need to be understood: first, it is distributed, which means that not every key will be on every server. In fact, a key should only be on one server, which allows the cluster to cache a large amount of data because the cache is the aggregate size of all servers. Second, the cache does not fill itself. While this may seem intuitive, the software using Memcached needs to ensure that after a read from the database, the data is stored in the cache for later retrieval. Entering data in the cache uses a *SET*, and retrieving data from the cache uses a *GET*. Lastly, there are several options to denote when data expires from the cache. Data can either have a time-to-live, or be present in the cache indefinitely. A caveat to using Memcached is that data will be removed from your cache if a server goes down as Memcached does not have persistent storage.

3.1.3.1 Versions and Scaling

There are several versions of Memcached: the current stable release is 1.4, and 1.6 is under development. Version 1.6 aims to fix scaling issues caused by running Memcached with a large number of threads. A detailed analysis is presented in [86]. Prior research has shown that Memcached saturates neither the network bandwidth, nor the memory bandwidth [54], due to inefficiencies in the TCP/IP stack. In this work, we distribute the work of the TCP/IP stack among many small cores to provide greater aggregate bandwidth while increasing the storage density. This is possible because 3D stacking provides higher bandwidth and a faster memory interface.

3.2 Related Work

Prior work has focused on increasing the efficiency or performance with respect to the RPS of Memcached systems, rather than density. As previously mentioned, we believe that density should be studied as a first class design constraint due to the high cost of scaling out data centers.

3.2.1 Characterizing Cloud Workloads

Ferdman et al. have demonstrated the mismatch between cloud workloads and modern out-of-order cores [22, 24]. Through their detailed analysis of scale-out workloads on modern cores, they discovered several important characteristics of these workloads: 1) Scale-out workloads suffer from high instruction cache miss rates, and large instruction caches and pre-fetchers, are inadequate; 2) instruction and memory-level parallelism are low, thus leaving the advanced out-of-order core underutilized; 3) the working set sizes exceed the capacity of the on-chip caches; 4) bandwidth utilization of scale-out workloads is low.

3.2.2 Efficient 3D-Stacked Servers

Prior work has shown that 3D stacking may be used for efficient server design. PicoServer [42] proposes using 3D stacking technology to design compact and efficient multi-core processors for use in tier 1 servers. The focus of the PicoServer is on energy efficiency—they show that by closely stacking low-power cores on top of DRAM they can remove complex cache hierarchies, and instead, add more low-power cores. The improved memory bandwidth and latency allow for adequate throughput and performance at a significant power savings. Nanostores [15] build on the PicoServer design and integrate Flash or Memristors into the stack. Both PicoServer and Nanostores could be used in a scale-out fashion to improve density, although this was not addressed in the work. This paper builds on their designs to address density, particularly in the context of Memcached.

More recently, Scale-Out Processors [59] were proposed as a processor for cloud computing workloads. The Scale-Out Processor design uses 3D stacked DRAM as a cache for external DRAM and implements a clever prefetching technique[36]. Our designs differ in that we only use the on-chip DRAM or Flash for storage with no external backing memory. This is possible because we share the Memcached data over several independent stacks in the same 1.5U box.

3.2.3 Non-Volatile Memory File Caching in Servers

In addition to Nanostores, several prior studies have proposed using non-volatile memory for energy-efficiency in servers [44, 46, 78]. They propose using non-volatile memory (NAND Flash and phase-change memory) for file caching in servers. The use of a programmable Flash memory controller, along with a sophisticated wear-leveling algorithm, allow for the effective use of non-volatile memory for file caching, while reducing idle power by an order of magnitude.

3.2.4 Super Dense Servers

Super Dense Servers [21] have been shown to provide greater performance and efficiency for CPU-bound electronic commerce workloads. However, for workloads that require a large amount of physical memory—such as Memcached—traditional servers provided a better solution. By utilizing state-of-the-art 3D-stacked memory technology, we overcome the limitation of Super Dense Servers by providing a very high level of memory density in our proposed server designs.

3.2.5 McDipper

Memcached has been used at Facebook for a wide range of applications, including MySQL look-aside buffers and photo serving. Using DRAM for these applications is relatively expensive, and for working sets that have very large footprints but moderate to low request rates, more efficient solutions are possible. Compared with DRAM, Flash solid-state drives provide up to 20× the capacity per server and still supports tens of thousands of operations per second. This prompted the creation of McDipper, a Flash-based cache server that is compatible with Memcached. McDipper has been in active use in production at Facebook for nearly a year to serve photos [25]. Our Iridium architecture targets these very large footprint workloads which have moderate to low request rates. We further extend their solution by using Toshiba's emerging 16-layer pipe-shaped bit cost scalable (p-BiCS) NAND Flash [41], which allows for density increases on the order of 5× compared to 3D-DRAM.

3.2.6 Enhancing the Scalability of Memcached

Wiggins and Langston [86] propose changes in Memcached 1.6 to remove bottlenecks that hinder performance when running on many core systems. They find that the locking structure used to control access to the hash table and to maintain LRU replacement for keys hinders Memcached when running with many threads. To mitigate the lock contention they use fine grain locks instead of a global lock. In addition, they modify the replacement algorithm to use a pseudo LRU algorithm, which they call Bags. Their proposed changes increase the bandwidth to greater than 3.1 MRPS, which is over 6× higher than an unmodified Memcached implementation.

3.2.7 TSSP

Lim et al. propose TSSP [54], which is an SoC including a Memcached accelerator to overcome the need for a large core in Memcached clusters. They find that, due to the network stack, Intel Atom cores would not be able to replace Intel Xeons in a standard cluster configuration. TSSP is able to offload all GET requests from the processor to the accelerator. The offload is accomplished by having a hash table stored in hardware and having a smart NIC that is able to forward GET requests to the accelerator. After data for a key is found, the hardware generates a response. Because little work needs to be done by software, an ARM Cortex-A9 is a suitable processor. The TSSP architecture achieves 17.63 KRPS/Watt.

3.2.8 Resource Contention in Distributed Hash Tables

distributed hash table (DHT) can suffer from issues that arise because there is not a uniform distribution of requests across resources. Keys in a key value store are assigned a resource by mapping a key onto a point in a circle. From this circle each node is assigned a portion of the circle, or arc. A server is responsible to store all data for keys that map onto their arc. Prior work dealing with resource contention in DHTs shows that increasing the number of nodes in the DHT reduces the probability of resource contention, because each node is responsible for a smaller arc [40, 80]. Typically, increasing the number of nodes has been accomplished by assigning one physical node to several virtual nodes. These virtual nodes are then distributed around the circle, which results in a more uniform utilization of resources.

Because we increase the number of physical cores with Mercury and Iridium, resource contention should be minimized.

3.2.9 TILEPro64

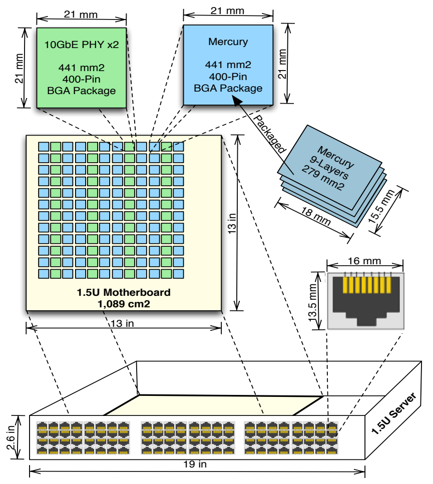
Berezecki et al. [9] focus on adapting Memcached to run on the *TILEPro64*. In this work they cite power consumption of data centers as an important component for the success of a web service. With this in mind, they aim to improve the efficiency of Memcached by running it on a TILEPro64. They compare their implementation to both Opteron and Xeon processors running Memcached, and report an RPS/W of 5.75 KRPS/W, which is an improvement of 2.85× and 2.43× respectively.

3.2.10 FAWN

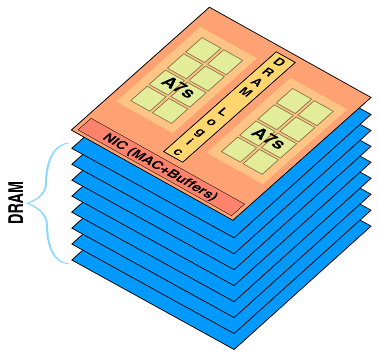
Andersen et al. design a new cluster architecture, called FAWN, for efficient, and massively parallel access to data [2]. They develop FAWN-KV—an implementation of their FAWN architecture for key-value stores that uses low-power embedded cores, a log-structured datastore, and Flash memory. With FAWN-KV they improve the efficiency of queries by two orders of magnitude over traditional disk-based systems. The FAWN system focuses on the key-value and filesystem design, whereas our work focuses on designing very dense servers.

3.3 Mercury and Iridium

The Mercury and Iridium architectures are constructed by stacking ARM Cortex-A7s, a 10GbE NIC, and either 4GB of DRAM or 19.8GB of Flash into a single 3D stack. The MAC unit of the NIC, which is located on the 3D stack, is capable of routing requests to the A7 cores. A conceptual representation is presented in Figure 3.2a. To evaluate the use of 3D stacks with key-value stores, we vary the number of cores per stack and measure the throughput and power efficiency for each server



(a) 1.5U server with 96 Mercury stacks



(b) The 3D-stacked Mercury architecture

Figure 3.2: A Mercury server and a single stack. A Mercury stack is made up of 8 DRAM layers, each 15.5mm×18mm, stacked with a logic layer containing the processing elements, DRAM peripheral logic, and NIC MAC & buffers. These stacks are then placed in a 400-pin BGA package. The 1.5U enclosure is limited to 96 Ethernet ports, each of which will be connected to a Mercury stack. Therefore, 96 Mercury stacks and 48 Dual NIC PHY chips are placed in the 1.5U enclosure. Due to space limitations a separate diagram of Iridium is omitted, however the high-level design is similar—the only difference being that we use a single layer of 3D-stacked Flash for the Iridium design.

configuration. We designate the different architectures as Mercury-n or Iridium-n, where n is the number of cores per stack. We estimate power and area requirements based off of the components listed in table 3.1.

3.3.1 Mercury

While the primary goal of Mercury and Iridium is to increase data storage density, this cannot be done at the expense of latency and bandwidth—the architecture would not be able to meet the service-level agreement (SLA) requirements that are typical of Memcached clusters. Thus we utilize low-power, in-order ARM Cortex-A7 cores in our 3D-stacked architecture, and as we will show in §3.5, we are able to service a majority of requests within the sub-millisecond range.

3.3.1.1 3D Stack

The proposed Mercury architecture relies on devices from Tezzeron's 3D-stacking process [81]. This process allows us to stack 8 memory layers on a logic die through finely-spaced (1.75μm pitch), low-power TSVs. The TSVs have a feedthrough capacitance of 2-3fF and a series resistance of $< 3\Omega$. This allows as much as 4GB of data in each individual stack.

The 4GB stack's logical organization is shown in Figure 3.3a. Each 4GB 3D chip consists of eight 512MB DRAM memory dies stacked on top of a logic die. The organization of the DRAM die is an extension [27] of Tezzaron's existing Octopus DRAM solution [82]. Each 3D stack has 16 128-bit data ports, with each port accessing an independent 256MB address space. Each address space is further subdivided into eight 32MB banks. Each bank, in turn, is physically organized as a 64×64 matrix of subarrays. Each subarray is a 256×256 arrangement of bit cells, and is 60μm×35μm.

Figure 3.3b shows the physical floor plan of each DRAM memory die and the logic die. The logic die is fabricated in a 28nm CMOS process and consists of address-decoding logic, global word line drivers, sense amplifiers, row buffers, error correction logic, and low-swing I/O logic with pads. Each memory die is partitioned into 16 ports with each port serving 1 of the 16 banks on a die. The memory die is fabricated in a 50nm DRAM process and consists of the DRAM subarrays along with some logic, such as local wordline drivers and pass-gate multiplexers.

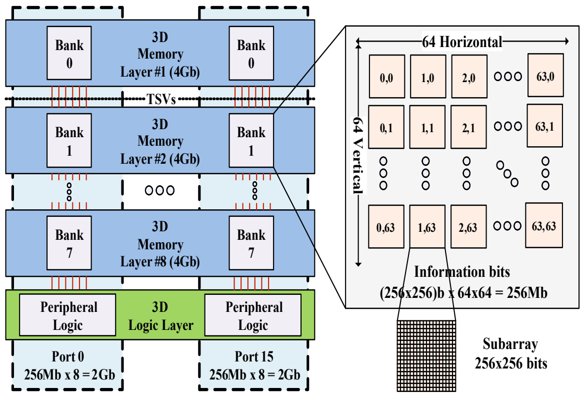
While there are more advanced DRAM processes (e.g. 20nm), TSV yield in existing 3D-stacked prototypes has only been proven up to the 50nm DRAM process node [47, 73]. All subarrays in a vertical stack share the same row buffer using TSVs, and at most one row of subarrays in a vertical stack can have its contents in the row buffer, which corresponds to a physical page. Assuming an 8kb page, a maximum of 2,048 pages can be simultaneously open per stack (128 8kb pages per bank $\times$ 16

Component	**Power (mW)**	**Area (mm^2)**
A7@1GHz	100	0.58
A15@1GHz	600	2.82
A15@1.5GHz	1,000	2.82
3D DRAM (4GB)	210 (per GB/s)	279.00
3D NAND Flash (19.8GB)	6 (per GB/s)	279.00
3D Stack NIC (MAC)	120	0.43
Physical NIC (PHY)	300	220.00

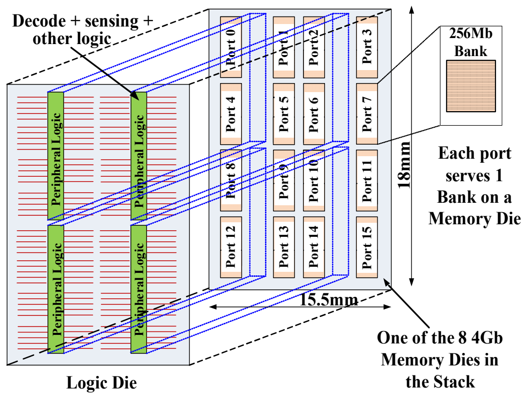
Table 3.1: Power and area for the components of a 3D stack.

DRAM	**BW (GB/s)**	**Capacity**
DDR3-1333 [73]	10.7	2GB
DDR4-2667 [73]	21.3	2GB
LPDDR3 (30nm) [5]	6.4	512MB
HMC I (3D-Stack) [73]	128.0	512MB
Wide I/O (3D-stack, 50nm) [47]	12.8	512MB
Tezzaron Octopus (3D-Stack)[82]	50.0	512MB
Future Tezzaron (3D-stack)[27]	100.0	4GB

Table 3.2: Comparison of 3D-stacked DRAM to DIMM packages.



(a) Logical organization of the 4GB 3D-stacked DRAM.



Decode + sensing + other logic
Peripheral Logic
Port 0
Port 15
256Mb Bank
Each port serves 1 Bank on a Memory Die
18mm
15.5mm
One of the 8 4Gb Memory Dies in the Stack
Logic Die

(b) Physical floor plan of the logical die and a 512MB memory die in the 3D stack. The top-down view of the stack shows that DRAM banks are arranged around spines of peripheral logic.

Figure 3.3: Logical organization and physical floorplan of the 3D DRAM.

banks per physical layer). The device provides a sustained bandwidth of 6.25GB/s per port (100GB/s total).

3.3.1.2 Address Space

The 3D stack DRAM has 16 ports for memory access, this segments the 4GB stack into 256MB chunks. Each core is allocated one or more ports for memory access, which prevents Memcached processes from overwriting each other's address range. If the Mercury or Iridium architectures increase past 16 cores, additional ports would need to be added, or cores would need to share ports.

3.3.1.3 Memory Access

Other studies have shown that small cores alone are not able to provide the needed bandwidth to be useful in key-value stores [54]. Mercury differs from this research because it is coupled with a memory interface that provides higher bandwidth at a lower latency through 3D integration. For comparison, Table 3.2 shows the bandwidth and capacity of several current and emerging memory technologies. Coupling cores on the 3D stack allows Mercury to forego using an L2 cache, which prior research has shown to be inefficient [54], and issue requests directly to memory. The 3D DRAM has a closed page latency of 11 cycles at 1GHz. By having a faster connection to memory we mitigate the cache thrashing by the networking stack reported in [54].

3.3.1.4 Request Routing

To simplify design and save power we do not use a router at the server level. Instead, the physical network port is tied directly to a 3D stack. This allows for each stack to act as a single node, or multiple nodes, without having contention from other stacks. At the stack, we base our design off of the integrated NIC on the Niagra-2 chip [7, 51]. The integrated NIC is capable of buffering a packet and then forwarding it to the correct core. Cores on the same stack will need to run Memcached on different TCP/IP ports. This simplifies request routing on the stack. The physical (PHY) portion of the 10GbE is based on the Broadcom design [12] and is not located on the stack.

3.3.2 Iridium

Due to the high cost of server real estate, physical density is first-class design constraint for key-value stores. Facebook has developed McDipper [25], which utilizes Flash memory to service low-request-rate transactions while improving density by up to 20×. McDipper has been in use for over a year to service photos at Facebook. With Iridium we target these low-request-rate applications and explore the tradeoff between physical density and throughput by replacing the DRAM in a Mercury stack with NAND Flash memory. This comes at the expense of throughput per GB of storage, but is still applicable to low-request-rate high-density Memcached clusters, such as Facebook's photo server.

3.3.2.1 3D Stacked Flash

McDipper reported an increase in density by 20× when moving from DRAM DIMMs to Solid-State drives. Because the Mercury design already improved density through 3D-DRAM we will not see as significant a gain. To quantify the improvement in the density of the stack we estimate our Flash density using the cell sizes of Toshiba's emerging pipe-shaped bit cost scalable (p-BiCS) NAND Flash [41]. Flash cells are smaller than DRAM cells, offering a 2.5× increase in density. Because p-BiCS has 16 layers of 3D Flash[1], as opposed to 8 layers for 3D-stacked DRAM, this leads to an overall 4.9× increase in density for Iridium stacks. For our access organization we maintain Mercury's 16 separate ports to DRAM by including 16 independent Flash controllers. The read/write latency values and energy numbers used for simulation are drawn from [28], which are conservative for 3D-stacked Flash. The power and area numbers for Flash are shown in table 3.1. In addition, because the Flash latency is much longer, an L2 cache is needed to hold the entire instruction footprint. Our results in §3.5.2 will confirm this assumption.

3.4 Methodology

The following sections describe our Memcached setup, as well as our simulation framework and power models.

[1]The 16 Flash layers are contained in a single monolithic layer of 3D Flash, and are not 3D die-stacked.

		1.5U Mercury Server						1.5U Iridium Server					
	Cores per Stack	1	2	4	8	16	32	1	2	4	8	16	32
A15 1.5GHz	Area(cm^2)	635	635	576	331	179	86	635	635	635	363	185	93
	Power(W)	449	569	749	750	750	720	328	449	690	740	737	730
	Density(GB)	384	384	348	200	108	52	1,901	1,901	1,901	1,089	554	277
	Max BW(GB/s)	27	55	99	114	123	118	1	3	6	7	7	7
A15 1GHz	Area(cm^2)	635	635	635	496	278	139	635	635	635	595	304	152
	Power(W)	401	473	618	745	742	728	281	353	498	750	740	728
	Density(GB)	384	384	384	300	168	84	1,901	1,901	1,901	1,782	911	455
	Max BW(GB/s)	27	54	108	169	189	189	2	3	6	11	11	11
A7 1GHz	Area(cm^2)	635	635	635	635	635	616	635	635	635	635	635	635
	Power(W)	341	353	378	429	529	749	221	233	258	309	410	612
	Density(GB)	384	384	384	384	384	371	1,901	1,901	1,901	1,901	1,901	1,901
	Max BW(GB/s)	19	37	75	149	299	578	1	3	6	12	22	44

Table 3.3: Power and area comparison for 1.5U maximum configurations. For each configuration we utilize the maximum number of stacks we can fit into a 1.5U server, which is 96, or until we reach our power budget of 750W. The power and bandwidth numbers are the maximum values we observed when servicing requests from 64B up to 1MB.

3.4.1 Memcached

For our experiments we utilize the latest stable version of Memcached cross-compiled for the ARM ISA. We modify our simulation infrastructure, which will be described in the next section, to collect timing information. We do this, as opposed to using library timing functions such as *gettimeofday()*, because they do not perturb the system under test. All experiments are run with a single Memcached thread.

For our client we use an in-house Java client that is based on the work by Whalin [85]. Because we measure performance from the server-side, the overhead of running Java is not included with the measurements.

3.4.2 Simulation Infrastructure

To calculate the request rates that Mercury and Iridium are able to achieve, we used the gem5 full-system simulator [10]. With gem5, we are able to model multiple networked systems with a full operating system, TCP/IP stack, and Ethernet devices. We use Ubuntu 11.04 along with version *2.6.38.8* of the Linux kernel for both the server and client systems. On the client side we use *Java SE 1.7.0_04-ea*.

While Mercury and Iridium both utilize ARM Cortex-A7 cores, we also explore the possibility of using the more aggressive, out-of-order Cortex-A15 core by modelling both cores in our simulation infrastructure. Simulations use a memory model with a memory latency varied from 10-100ns for DRAM and 10-20μs for Flash Reads. Our memory model represents a worst-case estimate as it assumes a closed-page latency for all requests. To measure the TPS we vary request size from 64B to 1MB, doubling request size at each iteration. We do not consider values greater than 1MB for the following reasons: 1) Memcached workloads tend to favor smaller size data; 2) prior work [54, 86, 4], against which we compare, present bandwidth per watt at data sizes of 64B and 128B; and, 3) requests that are 64KB or larger have to be split up into multiple TCP packets.

3.4.3 TPS calculation

To calculate the TPS we collect the RTT for a request, i.e., the total time it takes for a request to go from the client to the server, then back to the client. Because we use only a single thread for Memcached, the TPS is equal to the inverse of the RTT. The RTT for each request is obtained by dumping TCP/IP packet information from gem5's Ethernet device models. The packet trace is run through TShark [88] to parse out timing information. After timing information is obtained from gem5 for a single core, we apply linear scaling to calculate TPS at the stack and server level. Linear scaling is a reasonable approach in this context because each core on a stack is running a separate instance of Memcached. Running separate instances avoids the contention issue raised by the work of Wiggins and Langston[86]. For the Mercury-32 and Iridium-32 configurations we use the same approach, however we assume two cores per memory port (as mentioned previously we can fit a maximum of 16 memory ports on a stack), because Memcached has been shown to scale well for two threads [86].

3.4.4 Power Modeling

Table 3.1 has a breakdown of power per component, and Table 3.3 has the cumulative power totals for each configuration of Mercury and Iridium at their maximum sustainable bandwidths. To calculate the power of an individual stack we add together the power of the NIC, cores, and memory. The integrated 10GbE NIC is comprised of a MAC and buffers. The MAC power estimates come from the Niagra-2 design [7, 51], and the buffers are estimated from CACTI [68]. The ARM Cortex-A7 and Cortex-A15 power numbers are drawn from [30], and the 3D DRAM is calculated from a technical specification obtained from Tezzaron [27]. Because DRAM active power depends on the memory bandwidth being used, we must calculate the stack power for the maximum bandwidth that a given number of cores can produce. Similar calculations are used to obtain NAND Flash power, which use read and write energy values drawn from [28]. Finally, each stack requires an off-stack physical Ethernet port. Power numbers for the PHY are based on a Broadcom part [12].

3.4.4.1 Power Budget for 1.5U Mercury System

In calculating the power for the 1.5U server system we multiply the per stack power by the number of stacks. To determine how many stacks can fit in a 1.5U power budget, we start with a 750W power supply from HP [32]. First 160W is allotted for other components (disk, motherboard, etc.), after this we assume a conservative 20% margin for miscellaneous power and delivery losses in the server. This results in a maximum power of $(750 - 160) \times 0.8 = 472$W for Mercury or Iridium components.

3.4.4.2 TPS/Watt calculation

When calculating the maximum number of 3D stacks that fit in a system, we used the maximum memory bandwidth that Mercury and Iridium can produce. However, for proper TPS/Watt calculation, we estimate power by using the GB/s power consumption of DRAM and Flash at the request size we are testing.

3.4.5 Area

Area estimates for the 10GbE NIC were obtained by scaling the Niagra-2 MAC to 28nm and the buffers were obtained from CACTI. The area for an ARM Cortex-A7 chip in 28nm technology is taken from [30]. DRAM design estimates for the next generation Tezzaron Octopus DRAM were obtained from Tezzaron [27]. Given the available area on the logic die of the Tezzaron 3D-stack, we are able to fit >400 cores on a stack. However, the memory interface becomes saturated if there are ≥ 64 cores in a stack. We are further limited, by the number of DRAM ports in a stack, to 16 cores per stack unless cores share the memory controller. In each 1.5U server, multiple 3D stacks are used to increase the bandwidth and density. Once packaged into a 400-pin 21mm$\times$21mm ball grid array (BGA) package, each stack consumes 441mm^2. Each NIC PHY chip is also 441mm^2 and contains 2 10GbE PHYs/chip. If 77% of a 1.5U, 13in$\times$13in motherboard [52] is used for Mercury or Iridium stacks and associated PHYs, then the server can fit 128 Mercury or Iridium stacks. However, only 96 Ethernet ports can fit on the back of a 1.5U server [61]. Therefore, we cap the maximum number of stacks at 96.

3.4.6 Density

We define density to be the amount of DRAM that we can fit in the system. We then maximize density within three constraining factors: power, area, and network connections. As more cores are added to the system, the power requirement for both core power and DRAM bandwidth will increase, because of this there is a tradeoff between throughput and density. Each stack can fit 4GB of DRAM or 19.8GB of Flash and each server can fit up to 128 3D stacks, which gives a maximum density of 512GB of DRAM for Mercury or 2.4TB of Flash for Iridium. Each server can fit a maximum of 96 network connections, capping the number of stacks to 96 and density to 384GB of DRAM for Mercury or 1.9TB of Flash for Iridium.

3.5 Results

We evaluate several different 3D-stacked server configurations, which are differentiated based on the number (and type) of cores per stack, and on the type of memory used. DRAM-based configurations we call Mercury, while Flash-based configurations are called Iridium (because Iridium is more dense).

3.5.1 Request Breakdown

We first explore the different components of GET and SET requests. Figures 3.4 and 3.5 show a breakdown of execution time for both GET and SET requests respectively; execution is broken down into three components: hash computation time (*Hash Computation*), the time in metadata processing to find the memory location of the data (*Memcached*), and time spent in the network stack and data transfer (*Network Stack*). These experiments were run using a single A15 @1GHz, with a 2MB L2 cache, and DRAM with a latency of 10ns. We ran these experiments for various configurations, however the results were similar.

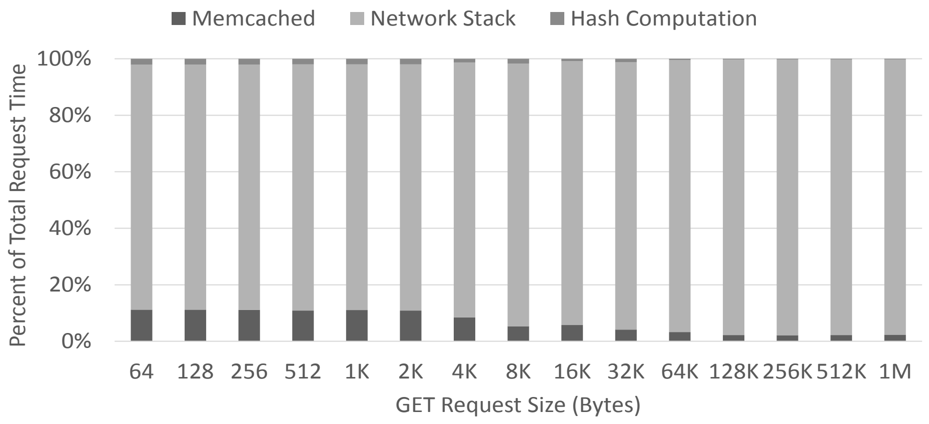


Figure 3.4: GET execution time.

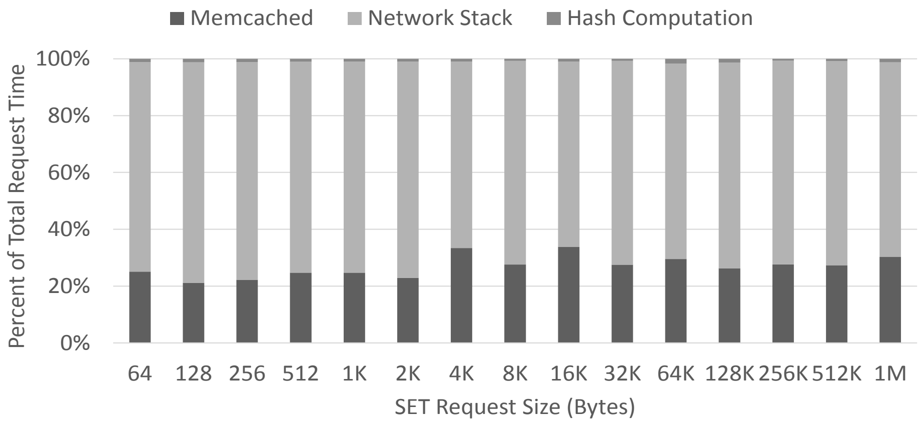


Figure 3.5: SET execution time.

3.5.1.1 GET and SET requests

Figure 3.4 shows the breakdown of execution during a GET request. For requests up to 4KB roughly 10% of time is spent in Memcached, 2-3% is spend in hash computation, and a vast majority of the time (87%) is spent in the network stack. At higher request sizes nearly all of the time is spent in the network stack, which is in agreement with prior research [54].

Figure 3.5 shows the breakdown of execution during a SET request. As expected, hash computation takes up the same amount of time for a SET request as it does for a GET request, however it represents a much smaller portion of the time: only around 1%. Also as expected, Memcached metadata manipulation takes up more computation for a SET request: up to 30% in some cases. Network processing is still the largest component at nearly 70% for some request sizes.

Because the network stack takes up a significant portion of the time, utilizing aggressive out-of-order cores is inefficient. As we will demonstrate, by closely integrating memory with many low-power cores, we can achieve a high-level of throughput much more efficiently than traditional servers.

3.5.2 3D-Stack Memory Access Latency Sensitivity

While the focus of this work is on improving density for Memcached servers, this cannot be provided at the expense of latencies that would violate the SLA. To explore the effect memory latency and CPU performance have on overall request RTT we measure the average TPS, which is the inverse of the average RTT for single-core Mercury and Iridium stacks. The lower the RTT the higher the TPS, thus a higher TPS indicates better overall performance for individual requests.

Figures 3.6, 3.7, 3.8, and 3.9, show the TPS sensitivity to memory latency, with respect to GET/SET request size, for a Mercury-1 stack. We evaluate a Mercury-1

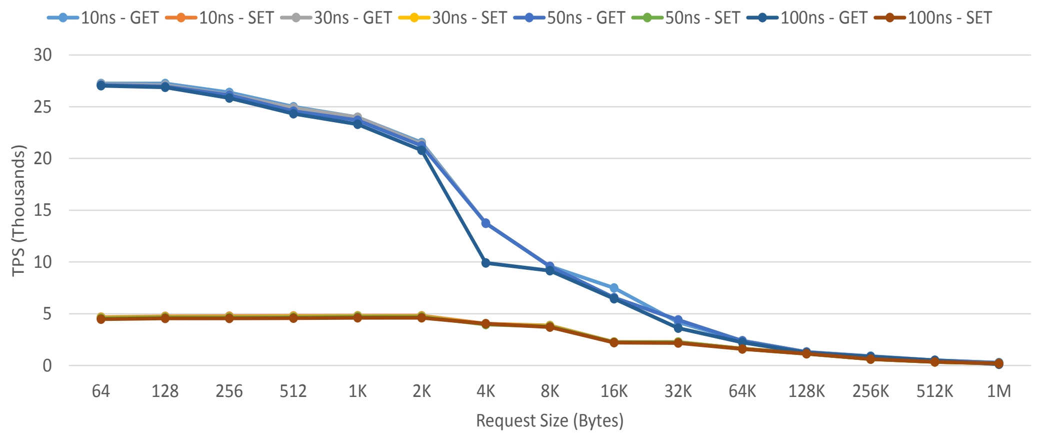


Figure 3.6: TPS for an A15-@1GHz-based Mercury stack with a 2MB L2 cache.

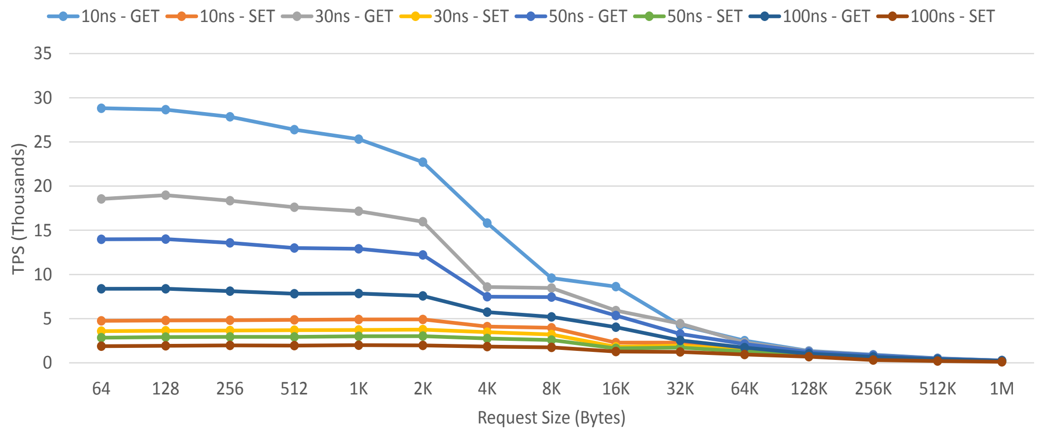


Figure 3.7: TPS for an A15-@1GHz-based Mercury stack with no L2 cache.

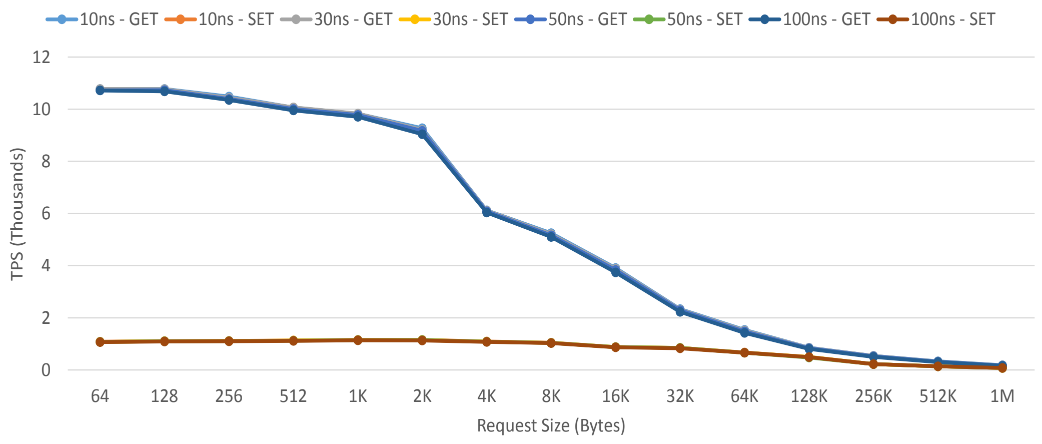


Figure 3.8: TPS for an A7-based Mercury stack with a 2MB L2 cache.

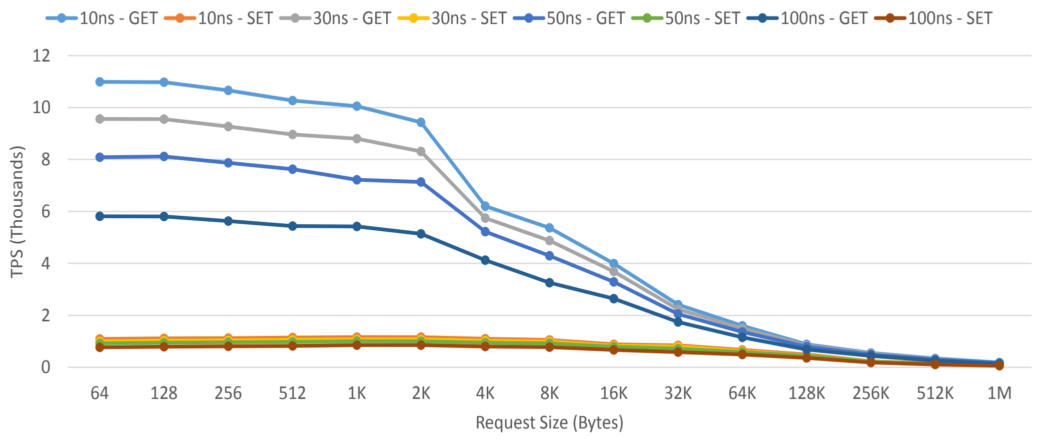


Figure 3.9: TPS for an A7-based Mercury stack with no L2 cache.

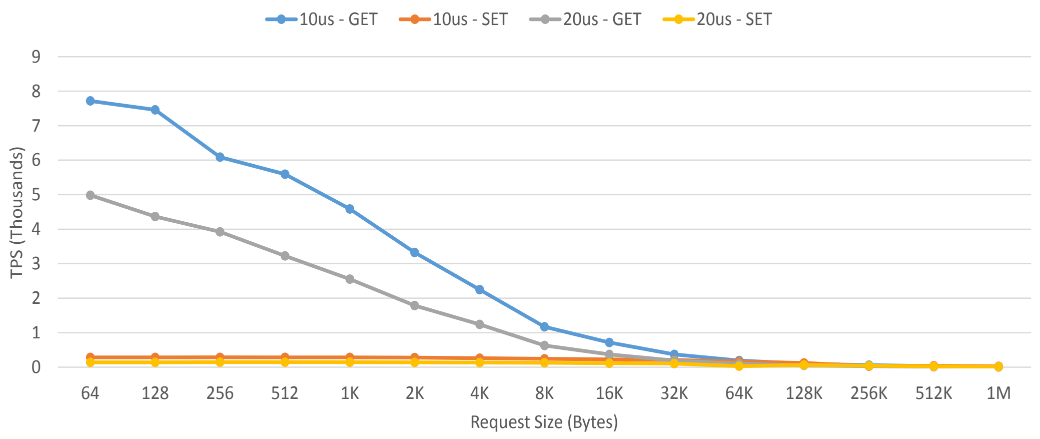


Figure 3.10: TPS for an A15-@1GHz-based Iridium stack with a 2MB L2 cache.

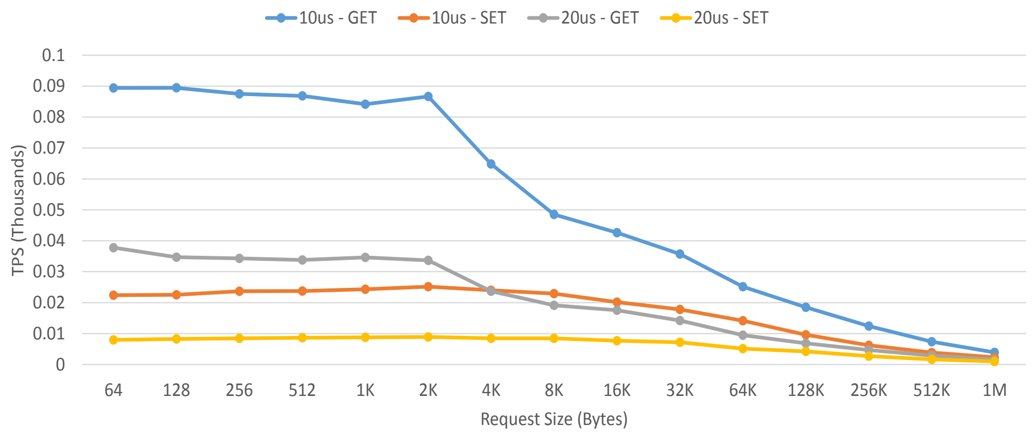


Figure 3.11: TPS for an A15-@1GHz-based Iridium stack with no L2 cache.

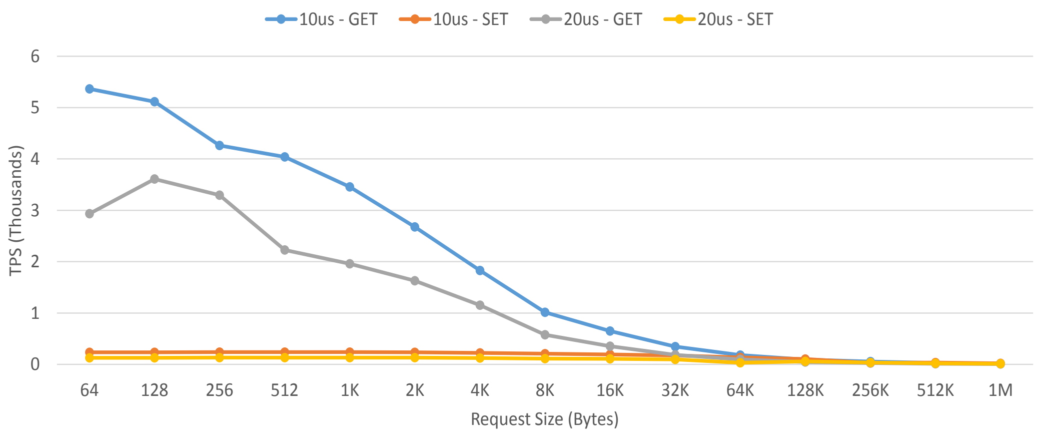


Figure 3.12: TPS for an A7-based Iridium stack with a 2MB L2 cache.

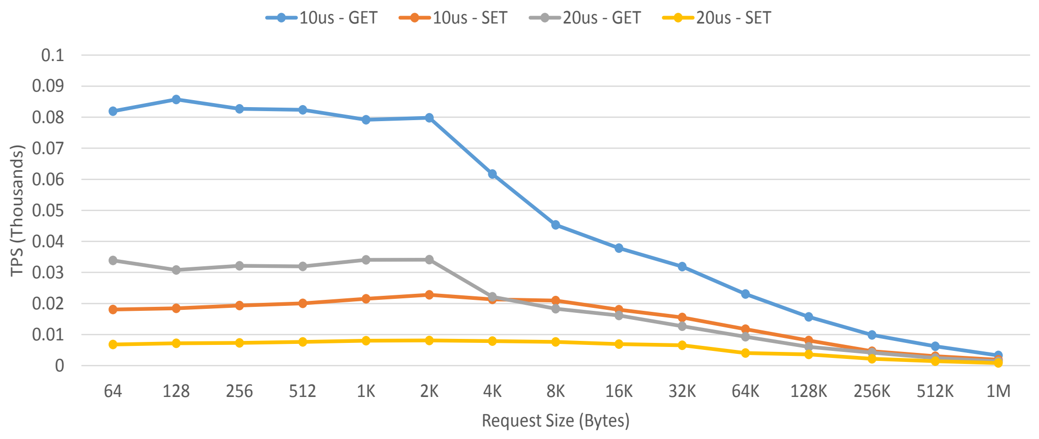


Figure 3.13: TPS for an A7-based Iridium stack with no L2 cache.

stack for an A15 and an A7, both with and without an L2 cache. We also explored using an A15 @1.5GHz, which is the current frequency limit of the A15, however we do not report these results because they are nearly identical to an A15 @1GHz. For each configuration we sweep across memory latencies of 10, 30, 50, and 100ns. Similarly, figures 3.10, 3.11, 3.12, and 3.13 demonstrate the throughput sensitivity to memory latency for an Iridium-1 stack. For Iridium-1 we sweep across Flash read latencies of 10 and 20μs; write latency is kept at 200 μs.

Figures 3.6 and 3.7 show the average TPS for an A15-based Mercury-1 stack with and without an L2 cache respectively. As can be seen, at a latency of 10ns the L2 provides no benefit, and may hinder performance. Because the DRAM is much faster than the core, the additional latency of a cache lookup, which typically has poor hit rates, degrades the average TPS. However, at the higher DRAM latencies the L2 cache significantly improves performance; while the stored values are typically not resident in the L2 cache, instructions and other metadata benefit from an L2 cache.

Similarly, figures 3.8 and 3.9 report average TPS for an A7-based Mercury-1 stack with and without and L2 cache. Because of the less aggressive core, the L2 cache makes less of a difference for an A7-based Mercury-1 stack. The A15-based Mercury-1 stack significantly outperforms an A7-based Mercury-1 stack by about 3× at the lower request sizes. If the L2 cache is removed, the A15 only outperforms the A7 by 1-2× at the lower request sizes.

Finally, figures 3.10,3.11, 3.12,and 3.13 report the average TPS for an Iridium-1 stack. Because of the relatively slow read and write latencies of Flash, an L2 cache is crucial for performance; removing the L2 cache yields average an average TPS below 100 for both the A7 and A15, which is not acceptable. However, with an L2 cache both the A15 and A7 can sustain an average of several thousand TPS for GET requests, with a bulk of the requests being serviced under 1ms. For SET requests the average TPS is below 1,000, however GET requests have been shown to make up a bulk of the requests for typical Memcached applications [4]. Because of Flash's relatively slow speed, the A15 outperforms an A7 by around 25% on average.

These results demonstrate the tradeoff between core and memory speed and performance. If very low response time is required for individual requests, faster memory and cores are desired. If, however, throughput and density are first-class constraints, then less aggressive cores and memory may be used without violating the SLA; as we will show in the next sections, density and efficiency may be vastly improved.

3.5.3 Density and Throughput

We define the density of a stack to be the total amount of memory it contains. Figures 3.14 and 3.15 illustrate the tradeoff between density and total throughput for different Mercury and Iridium configurations. Throughput is measured as the average TPS for 64B GET requests; prior works have shown that small GET requests are the most frequent requests in Memcached applications, and base their evaluations on such requests [86, 4]. Each configuration is labelled Mercury-n or Iridium-n, where n is the number of cores per stack. For all Mercury configurations we use a DRAM latency of 10ns, and for all Iridium configurations we use Flash read and write latencies of 10 and 200μs respectively. In each configuration the core contains a 2MB L2 cache. table 3.3 lists the details of each separate Mercury and Iridium configuration.

The core power (listed in table 3.1) is the limiting factor when determining how many total stacks our server architecture can support. As figures 3.14 and 3.15 show, the A15's higher power consumption severely limits the number of stacks that can fit into our power budget. At 8 cores per stack we see a sharp decline in density, while performance levels off. The A15's peak efficiency comes at 1GHz for both Mercury-8 and Iridium-8 stacks, where we are able to fit 75 stacks (600 cores) and 90 stacks (720 cores) respectively; Mercury-8 can sustain an average of 17.29 million TPS with 300GB of memory, while Iridium-8 can sustain an average of 5.45 million TPS with

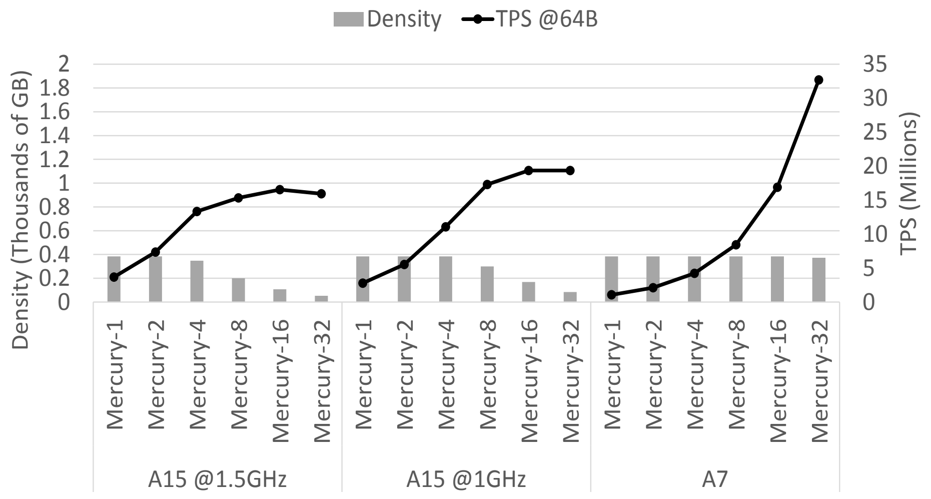


Figure 3.14: Density and throughput for Mercury stacks servicing 64B GET requests.

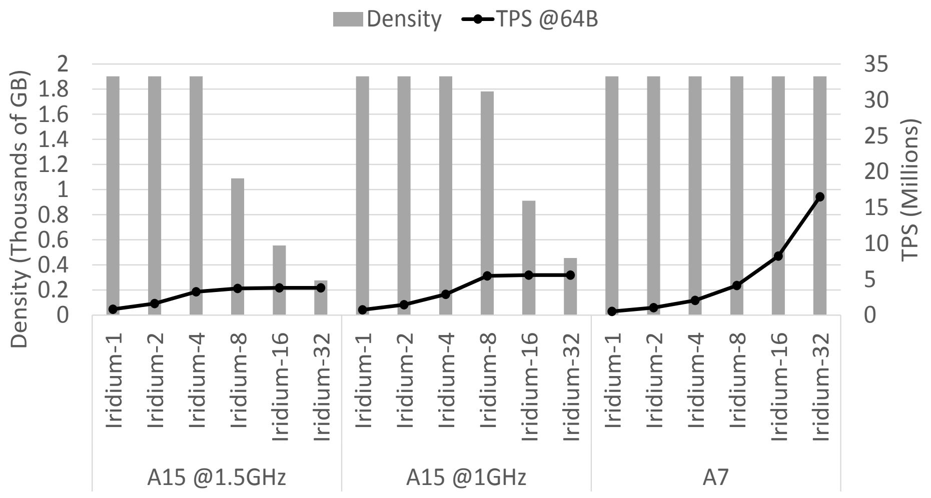


Figure 3.15: Density and throughput for Iridium stacks servicing 64B GET requests.

approximately 2TB of memory.

The A7's relatively low power allows us to fit nearly the maximum number of stacks into our server, even at 32 cores per stack. Because of this, the A7-based Mercury and Iridium designs are able to provide significantly higher performance and density than their A15-based counterparts. Mercury-32 can sustain an average TPS of 32.7 million with 372GB of memory, while Iridium-32 can sustain an average TPS of 16.48 with approximately 2TB of memory.

The A7 provides the most efficient implementation for both Mercury and Iridium

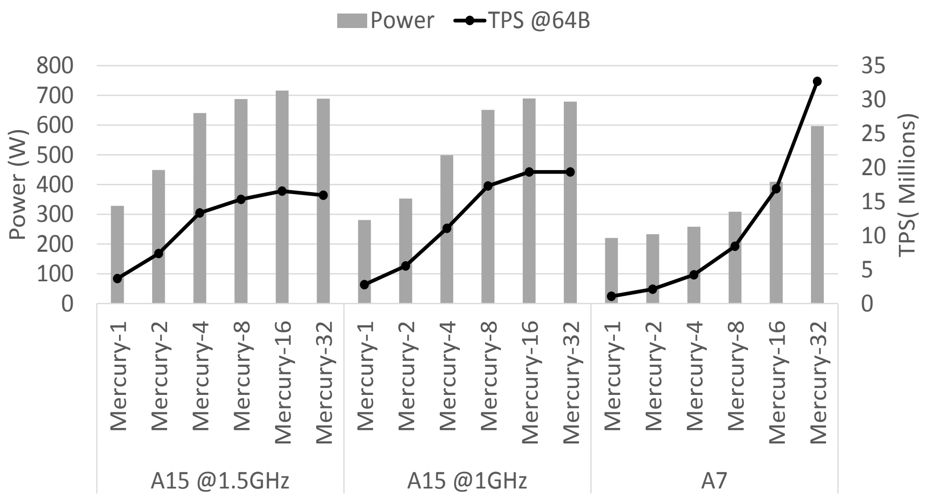


Figure 3.16: Power and throughput for Mercury stacks servicing 64B GET requests.

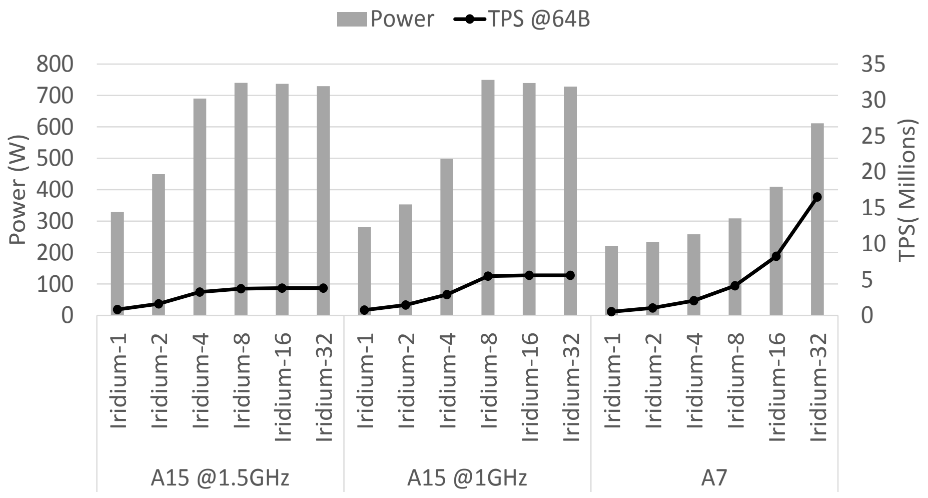


Figure 3.17: Power and throughput for Iridium stacks servicing 64B GET requests.

stacks, however Mercury-32 provides around 2× more throughput when compared to Iridium-32; Iridium-32, on the other hand, provides nearly 5× more density. If performance is a primary concern Mercury-32 is the best choice. If high density is needed, Iridium-32 provides the most memory and is still able to satisfy the SLA requirements.

3.5.4 Power and Throughput

The power and throughput tradeoff is shown in figures 3.16 and 3.17. Again, we measure the throughput of our system while servicing 64B GET requests. Because we are limited to a maximum of 96 stacks for a single server, the configurations that contain 1, 2, or 4 cores are well under our maximum power budget of 750W, however as we add more cores per stack we come close to saturating our power budget. As power becomes a constraint, the number of stacks we are able to fit into our power budget is reduced. Thus, there is a tradeoff between total throughput and overall power. We seek to maximize throughput while staying within our power budget, therefore we always opt for a system with the maximum number of stacks if possible.

Figure 3.16 shows that, for Mercury, the A15's power quickly becomes prohibitive, limiting the number of cores we can fit into a server given our 750W power budget. The best A15 configuration is a Mercury-16 system that uses A15s @1GHz. An average of 19.36 million TPS can be sustained at a power of 678W. The max throughput for a Mercury-32 system using A15s @1GHz uses slightly less power than the Mercury-16 system, while delivering nearly the same throughput, because less stacks are used. Using A7s we are able to fit nearly the maximum number of stacks, while staying well below our power budget. A Mercury-32 system using A7s is the most efficient design, delivering 32.7 million TPS at a power of 597W.

For Iridium, the A15 is even less efficient, as seen in figure 3.17—the A15's extra power does not help performance because it often waits on memory. The throughput

		Mercury			Iridium			Memcached			TSSP
	Version	n=8	n=16	n=32	n=8	n=16	n=32	1.4	1.6	Bags	—
1.5U Server @64B Request	Stacks	96	96	93	96	96	96	1	1	1	1
	Cores	768	1,536	2,976	768	1,536	3,072	6	4	16	1
	Memory(GB)	384	384	372	**1,901**	**1,901**	**1,901**	12	128	128	8
	Power(W)	309	410	597	309	410	611	143	159	285	16
	TPS(millions)	8.44	16.88	32.70	4.12	8.24	16.49	0.41	0.52	3.15	0.28
	TPS(thousands)/Watt	27.33	41.21	**54.77**	13.35	20.13	26.98	2.9	3.29	11.1	17.6
	TPS(thousands)/GB	21.98	43.96	**87.91**	2.17	4.34	8.67	34.2	4.1	24.6	35.3
	Bandwidth(GB/s)	0.54	1.08	2.09	0.26	0.53	1.06	0.03	0.03	0.20	0.04

Table 3.4: Comparison of Mercury and Iridium to prior art. We compare several versions of Mercury and Iridium (recall n is the number of cores per stack) to different versions of Memcached running on a state-of-the-art server, as well as TSSP. The bold values represent the highest density (GB), efficiency (TPS/W), and accessibility (TPS/GB).

for an A7-based Iridium-32 system is half that of an A7-based Mercury-32 system at roughly the same power budget. However, as noted above the Iridium-32 system provides 5× more density. The power of Iridium is slightly higher than Mercury because the relatively low power of Flash allows for more stacks.

3.5.5 Cooling

The TDP of a Mercury-32 server is 597W (the same as today's 1.5U systems) and is spread across all 96 stacks; in contrast to a conventional server design where all the power is concentrated on a few chips. This yields a TDP for an individual stack of 6.2W. Thus, we expect the power of each Mercury chip to be within the capabilities of passive cooling, and an active fan in the 1.5U box can be used to extract the heat. Prior work on the thermal characterization of cloud workloads also supports this notion [65].

3.5.6 Comparison to Prior Work

To gauge the performance of Mercury and Iridium we compare their efficiency, density, and performance to modern versions of Memcached running on a state-of-the-art server. Table 3.4 lists the pertinent metrics for the best performing Mercury and Iridium configurations, and compares them to prior art. As can be seen in this table both Mercury and Iridium provide more throughput, 10× and 5.2× more than Bags respectively, primarily due to the massive number of cores they can support. At the same time, the use of low-power cores and integrated memory provides greater efficiency: 4.9× higher TPS/W for Mercury and 2.4× more for Iridium. Because of the speed of memory used in Mercury systems, it can even make better use of its density yielding an average TPS/GB that is 3.4× higher than Bags. Iridium has 2.8× less TPS/GB on average due to its much higher density. Overall, Mercury and Iridium provide 2.9 × and 14.8× more density on average, while still servicing a majority of requests within the sub-millisecond range.

The table also reports the same metrics for TSSP, which is an accelerator for Memcached. While this work aims to improve the efficiency of Memcached by using specialized hardware, the 3D-stacked Mercury and Iridium architectures are able to provide 3× and 1.5× more TPS/W respectively.

3.6 Conclusion

Distributed, in-memory key-value stores, such as Memcached, are so widely used by many large web companies that approximately 25% of their servers are devoted solely to key-value stores. Data centers are expensive, a fact which requires that each unit be used as efficiently as possible. While previous works have recognized the importance of key-value stores, and the fact that they are inefficient when run on commodity hardware, the approach has typically been to try to improve the performance and efficiency of existing server systems.

In this work we propose using state-of-the-art 3D stacking techniques to develop two highly-integrated server architectures that are not only able to allow low-power, embedded CPUs to be used without sacrificing bandwidth and performance, but are also able to drastically improve density. This is a crucial component to keeping the cost of ownership for data centers down. Through our detailed simulations, we show that, by using 3D stacked DRAM (Mercury), density may be improved by 2.9×, efficiency by 4.9×, TPS by 10×, and TPS/GB by 3.5× over a current state-of-the-art server running an optimized version of Memcached. By replacing the DRAM with Flash our Iridium architecture can service moderate to low request rate servers with even better density while maintaining SLA requirements for the bulk of requests. Iridium improves density by 14×, efficiency by 2.4×, TPS by 5.2×, with only a 2.8× reduction in TPS/GB compared to current Memcached servers.

CHAPTER IV

Integrated Networking for Dense Servers

4.1 Introduction

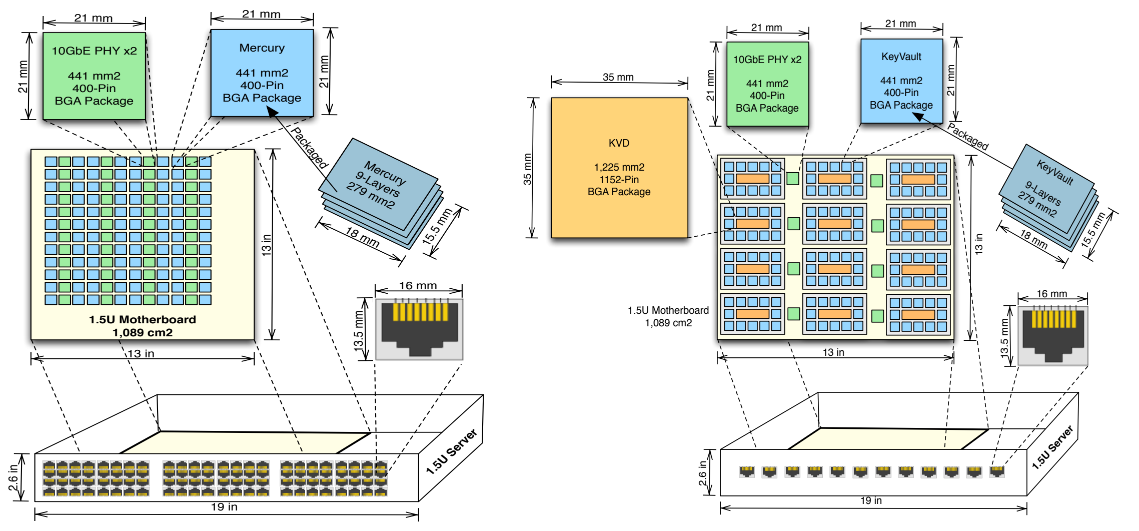
Internet services operate on massive data sets and are responsible for servicing many real-time requests to small, disparate pieces of data. Servicing these requests from the back-end databases does not scale well; fortunately the size and popularity of requested objects exhibit power-law distributions [3, 17], which makes them suitable for in-memory caching. As we enter the peta-scale era, key-value stores will become one of the foremost *scale-out* workloads used to improve the scalability of internet services. As such, vast data center resources will be dedicated solely to key-value stores in order to ensure that QoS guarantees are met.

Key-value stores are currently deployed on commodity off-the-shelf hardware, which is not ideal due to numerous inefficiencies caused by a mismatch with the application domain. The highly distributed nature of key-value stores requires frequent remote memory accesses, which can be several orders of magnitude slower than local memory when nodes are connected using commodity networking technology. Several prior works have shown that key-value store applications do not fully utilize high-end OoO CPUs—exhibiting poor cache hit rates, and high branch mis-prediction rates—and do not come close to saturating available memory or network bandwidth [2, 22, 23, 56]. The primary cause of this inefficiency is the overhead associated with processing individual network packets, which typically carry small payloads on the order of kilobytes. The large code footprints associated with the networking stack and operating system code cause a bottleneck in the instruction fetch portion of the CPU; and complex, off-chip NICs cause slow data transfers.

As the bandwidth of network interfaces approaches 1Tb/s [87]—matching the available bandwidth of the CPU's memory controller—the problem of network packet processing will be exacerbated. The overhead of relying on the CPU and OS to process packets, and get data off chip, will be the limiting factor to fully utilizing available network bandwidth. To ensure QoS requirements are met, on-chip networking hardware with DMA capabilities—along with lightweight networking protocols—will be crucial.

In this work we design simple, integrated networking hardware that allows for a low-overhead—yet general purpose and powerful—networking protocol. We propose KeyVault, building upon the Mercury server design [29], extending it to include simple, zero-copy NICs along with an integrated switch fabric and a lightweight communication protocol. In summary, we make the following contributions:

- We make the observation that, in order to achieve maximum performance, it is not necessary to bypass the CPU and OS completely when processing requests. Performing zero-copy packet transmission is enough to outperform solutions that completely bypass the CPU.
- Design an integrated NIC that allows for true zero-copy packet transmission.
- Specify a networking protocol that makes full use of the integrated NICs, thereby reducing the RTT of requests, and improving the max sustainable RPS.
- Design an integrated switching fabric that allows the network links to be shared amongst several stacks, thereby improving the network link utilization.



(a) Mercury (b) KeyVault

Figure 4.1: 1.5U Server configuration for Mercury and KeyVault servers. By reducing the need for complex networking hardware, a KeyVault server can fit more memory and compute into a single 1.5U server box than a Mercury server while improving throughput and latency.

- Provide a detailed design space exploration of various 1.5U KeyVault server designs, contrasted with prior works.

The rest of the paper is organized as follows: we motivate the need for physical density in §4.2; we then describe how our networking hardware and communication protocol improve the efficiency and performance of dense servers in §4.3 and §4.4; in §4.5 we provide details about our experimental methodology and evaluation; we discuss and analyze the results of our evaluation in §4.6; finally, we discuss prior work in §4.7 and conclude in §4.8.

4.2 A Case for Physically Dense Servers

In this section we describe some of the characteristics of modern data centers that highlight the need for dense server architectures.

4.2.1 Warehouse-Scale Computing

Figure 1.2 shows the three-tiered structure of modern internet services. In the first tier requests are routed by a DNS to a front-end server, which is responsible for HTTP caching, load balancing, etc. The load balancer schedules requests to be serviced by the application servers—which generate content—based on queue length, quality of placement, and the resource requirements of the request. All content is stored in the back-end database, however alongside the database there is typically an in-memory caching layer that stores frequently accessed objects in memory. As the volume of data in the cloud increases, there is a growing trend towards distributing large data sets across many servers and pushing the compute—e.g., database queries, key-value GETs, etc.—to the data. The highly distributed nature of these applications requires that most data accesses must be done over the data center's network, which is a source of significant delay [79].

Data centers are provisioned to meet peak demand, ensure low variability for response times, and provide fault tolerance. Current data centers employ commodity hardware because of its cost-effective performance. This leads to data center designs that push the limits of power supplies, warehouse space, and cooling, without fully utilizing the available compute, storage, or communication resources; data centers typically operate between 10-50% utilization on average [8].

4.3 KeyVault

In this section we describe the Mercury server architecture as described in [29], and describe how KeyVault extends that work to increase throughput, reduce latency, and improve efficiency of the network and memory bandwidth.

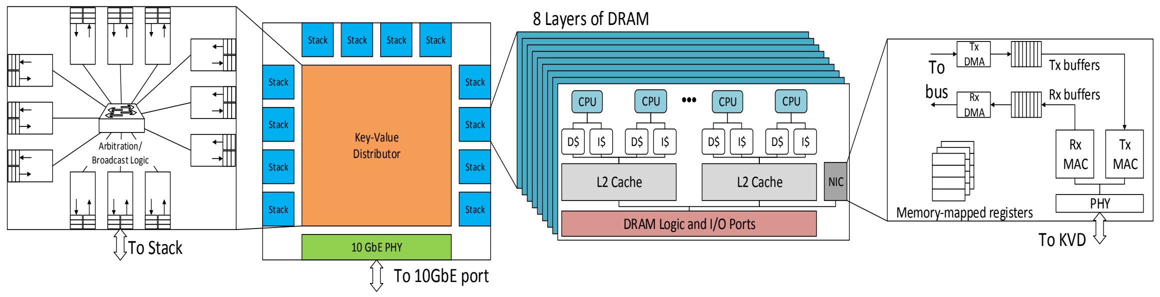


Figure 4.2: The components of a KeyVault server in detail.

4.3.1 Mercury

In [29], the authors propose several server designs—using 3D-stacked memory and low-power CPUs—in order to improve the amount of memory density and throughput achievable by a single 1.5U key-value store server. They perform a design space exploration of two server architectures: Mercury (DRAM-based) and *Iridium* (FLASH-based). Their servers are able to fit 96 stacks, each of which contains up to 32 CPUs 3D-stacked DRAM; or a single layer of 3D FLASH in the case of Iridium. Figure 4.1a shows the layout of a single 1.5U Mercury server. Their results ultimately show that by using low-power, in-order CPUs along with 3D-stacked DRAM, memory density can be improved. Moreover, it can achieve a high overall request rate while meeting a strict SLA.

While their proposed architectures improve the overall throughput, memory density, and power efficiency of a 1.5U key-value store server, they do not address the issue of high network overhead, which leaves the 96 10GbE links their servers require highly underutilized. Their work also does not address the need for expensive, top-of-rack switches required to facilitate the 96 Ethernet ports on their 1.5U box. The network overhead also limits the potential latency and throughput improvements that are available by using fast, 3D-stacked memory.

4.3.2 3D-Stack Technology

The KeyVault architecture relies on the same 3D-stacking process as Mercury [82]. This process allows us to stack 8 memory layers on a logic die through finely-spaced (1.75μm pitch), low-power TSVs. The TSVs have a feedthrough capacitance of 2-3fF and a series resistance of $< 3\Omega$. This allows up to 4GB of data in each individual stack.

Each 4GB 3D chip consists of eight 512MB DRAM memory dies stacked on top of a logic die. The organization of the DRAM die is an extension [26] of Tezzaron's existing Octopus DRAM solution [82]. Each 3D stack has 16 128-bit data ports, with each port accessing an independent 256MB address space. Each address space is further subdivided into eight 32MB banks. Each bank, in turn, is physically organized as a 64×64 matrix of subarrays. Each subarray is a 256×256 arrangement of bit cells, and is 60μm×35μm.

The logic die is fabricated in a 28nm CMOS process and consists of address-decoding logic, global word line drivers, sense amplifiers, row buffers, error correction logic, and low-swing I/O logic with pads. Each memory die is partitioned into 16 ports with each port serving 1 of the 16 banks on a die. The memory die is fabricated in a 50nm DRAM process and consists of the DRAM subarrays along with some logic, such as local wordline drivers and pass-gate multiplexers.

While there are more advanced DRAM processes (e.g. 20nm), TSV yield in existing 3D-stacked prototypes has only been proven up to the 50nm DRAM process node [47, 73]. All subarrays in a vertical stack share the same row buffer using TSVs, and at most one row of subarrays in a vertical stack can have its contents in the row buffer, which corresponds to a physical page. Assuming an 8kb page, a maximum of 2,048 pages can be simultaneously open per stack (128 8kb pages per bank $\times$ 16 banks per physical layer). The device provides a sustained bandwidth of 6.25GB/s per port (100GB/s total).

4.3.3 KeyVault

The KeyVault architecture builds off the Mercury design—using the same 3D-stacked DRAM technology—yet aims to eliminate the inefficiencies present due to high network overheads. KeyVault is able to realize the full potential of the 10GbE links and 3D-stacked memory, while maintaining all of the density benefits achieved by Mercury. A detailed diagram of a 1.5U KeyVault server is shown in figure 4.1b.

We remove the need for 96 10GbE ports by including a *key-value distributor (KVD)*—an integrated switching fabric that can connect a single 10GbE port to a cluster of stacks. By doing so we not only remove the need for many expensive networking components, we also improve the utilization of the available network links. Another key component of the KeyVault architecture is the inclusion of an integrated NIC that is connected directly to the cache hierarchy, and is capable of zero-copy packet transfer. A simplified communication protocol, discussed in section 4.4, reduces per-packet processing overhead, thereby reducing overall request latency.

The individual KVDs—each with its own MAC and IP addresses—are connected to a single 10GbE port and are able to directly communicate with the stacks in a single cluster. Each KVD can communicate with other KVDs to allow stacks in separate clusters to communicate with each other. We use 10GbE technology in our study for simplicity, however our designs do not rely on this and could be seamlessly integrated with other communication links, such as InfiniBand.

4.3.3.1 Key-Value Distributor

The KVD contains a reliable, full-duplex switching fabric. The switch fabric is implemented as an n×n crossbar, where n is the number of connected stacks, and buffers packets at the input and output ports. Because the KVD's ports are hardwired to stacks, there is no need for costly TCAM lookups to determine the destination port;

instead stacks are statically assigned stack IDs, and the KVD performs simple packet inspection on the network layer packets in order to route packets. The NIC and KVDs operate on standard Ethernet frames.

The KVD contains two primary engines that are responsible for forwarding packets to their destination: a *broadcast engine* and a *forwarding engine*. The broadcast engine is responsible for packet replication and ensuring that a packet is broadcast to all connected ports. A special stack ID is designated in order to specify that a broadcast should be performed. The forwarding engine sends packets from the receive (Rx) queues of the input ports, to the transmit (Tx) queues of the destination port. A simple least-recently-granted (LRG) arbitration policy is used to handle requests that compete for a destination port.

4.3.3.2 Zero-Copy NIC Design

The on-stack NICs and are inspired by previous work on highly integrated NICs [11, 35]. The NICs comprise Rx and Tx DMA engines, a set of memory-mapped registers, and Rx and Tx buffers. It connects directly to the cache hierarchy, however it does not respond to snoop requests. This allows zero-copy placement of the NIC's payload directly into an application's user-space buffers, eschewing the need for complex DMA descriptor rings. The NIC has access to the system MMU and TLB to perform address translation, and the DMA engines operate on physical addresses.

In order to simplify the communication protocol and improve throughput, our NIC virtualizes its registers, buffers, and DMA engines on a per-connection basis. Each open connection to the NIC has a unique connection ID, which is 16 bits in length, and is used to identify the open file descriptor associated with a particular connection. The status register contains flags regarding whether or not the device is available to receive or transmit data: *RXREADY* and *TXREADY*, as well as flags to indicate

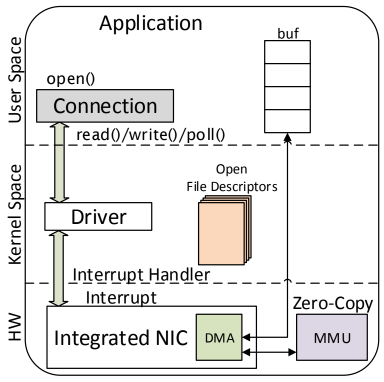


Figure 4.3: The HW/SW interface of the integrated NIC.

that a receive or transmit operation has completed: *RXDONE* and *TXDONE*. These flags are primarily used by the `poll()` system call. There is also a packet offset field: *PKTOFFSET* that specifies the offset into the NICs buffer if a packet was not completely read during a `read()` system call. Finally, there are registers that hold the starting address to be read from or written to: *RXADDR* and *TXADDR* respectively. The command register contains two flags: *RXGO* and *TXGO*, which are used to start a receive or transmit operation respectively. The command register also contains a field that specifies how many bytes are to be read or written. The NIC can generate interrupts when there is data present for reading, or if buffer space becomes available after a transmit completes.

4.4 Communication Protocol and Software Layer

User applications send and receive packets via a lightweight software stack that implements a low-overhead communication protocol. The software library interacts with the integrated NIC through its driver, which provides support for the standard file operations: `open()`, `read()`, `write()`, `poll()`, etc.

4.4.1 Device Driver

User applications open connections to the integrated NIC via a lightweight networking API, which uses the `open()` system call as a backend. Each open connection creates a new file descriptor on the integrated NIC—e.g., `/dev/kv_accel/connection_id`—and is associated with its own unique connection ID, which indicates to the driver which file descriptor a connection belongs to. Each connection has its own set of virtual registers within the NIC.

Figure 4.3 shows the overall software flow for sending and receiving packets via the integrated NIC. Memcached uses libevent, which relies on the `poll()` system call, to perform non-blocking I/O. When the Memcached client or server start, they register an event loop that polls on an open connection to determine whether or not it is available for reading or writing. The device driver reads the status register of the NIC for a particular connection to determine if it is indeed available, and if not, puts the polling thread on a wait queue. Once the device's buffers have data available for reading, or space is freed up in the case of a write, the device triggers an interrupt. The driver's interrupt handler will then wake up any process waiting on that connection, which triggers a libevent callback to be executed. The callback then reads or writes data—via `read()` or `write()` system calls—and processes any commands received. On a read or write the driver simply sends the address of the user space buffer to be read from or written to, and it not responsible for copying data to or from kernel space buffers. It should be noted that polling is not necessary to send or receive packets via the integrated NIC; polling just happens to be the way Memcached implements non-blocking I/O.

Because our NICs are only directly accessing per-connection buffers—and not actually touching the applications internal data structures—all synchronization can be handled by the software, which simplifies our zero-copy implementation.

Component	**Power (mW)**	**Area (mm^2)**	**Parameters**
A15-like CPU	1,000	2.82	3-wide, 1.5GHz, 32KB L1, 2MB L2
3D DRAM	210	279	4GB, 10ns R/W
NIC	120	0.43	128KB Rx/Tx buffers
KVD	819	32	1Gbps links, 12 ports, 128KB Rx/Tx buffers
PHY	300	220	10GbE

Table 4.1: Per component descriptions for a KeyVault cluster.

4.4.2 Communication Protocol

KeyVault servers communicate over a simple, connection-based protcol, which primarily touches the data link, network, and transport layers. The communication protocol combines aspects from both RDMA over Infiniband as well as traditional TCP/IP, however there are differences from both as well. In particular, the software layer assumes that packets are delivered over a reliable switching fabric—this means that the switching fabric does not drop packets due to full queues. To accomplish this the fabric must provide some sort of lossless flow control e.g., credit-based flow control where packets may only be sent over links if it has enough credits to do so; the number of credits available are based on the available queue space at the other end of the link. The software layer also creates virtual point-to-point connections—similar to Infiniband—between endpoints, this ensures that packets will not be reordered with respect to packets in the same virtual channel. Packets may be reordered over the physical link with respect to packets from other virtual channels however. The only errors that can occur are bit errors due to the physical link corrupting bits, which is extremely rare. To correct for this an error correcting code is used and the software transport layer handles the error checking and resending of packets, similar to traditional TCP/IP.

4.4.2.1 Data Link Layer

Packets are encapsulated in standard Ethernet frames and delivered over our reliable switching fabric, meaning that packets are not dropped due to full queues. A sliding window protocol is used for flow control. When the hardware buffers become full for a given connection, the process to which the connection belongs waits until space becomes free. Packet loss can only occur when bit errors occur due to hardware failures, which are rare. Error checking and packet retransmission is handled at the transport layer.

4.4.2.2 Network Layer

Once a packet is routed to a KVD via its IP and MAC addresses, it is then forwarded to a stack based on stack ID. The network layer packet's header contains the source and destination stack IDs. We use 4 bits for each field— we cap the number of connected stacks at 12—and addresses 0xE and 0xF are used to indicated broadcasts and multicasts respectively. As previously mentioned, the KVDs inspect packets to retrieve the associated stack IDs so they may directly forward packet to their destination stack.

4.4.2.3 Transport Layer

The primary fields in the transport layer's header are the source and destination connection IDs—each of which is 16 bits long—and a 32 bit request ID that is associated with outstanding requests at a sender connection. The header also contains a 32 bit checksum. Once the packet reaches the NIC of its destination stack, the on-stack NIC inspects the packet once again in order to determine the connection for which it is bound. The packet is placed into the hardware buffer associated with the destination connection. If necessary, an interrupt is generated to let the driver know that a connection has received a packet.

At the destination node, the networking software performs a checksum on the packet and sends an appropriate response to the sending connection. In the case of a bit error, a reply is sent to the source node requesting a retransmission of the packet. The destination for the reply packet is determined based on the sender's connection and stack IDs, as well as its IP and MAC addresses. At the sender , each request has an associated request ID, which is used to determine which request a response is acknowledging. The destination's stack and connection ID are determined by performing a simple hash on the request's key.

4.5 Experimental Methodology

We evaluate an entire KeyVault system—and replicate the Mercury design—using detailed full-system simulation. Client and Server machines are modeled.

4.5.1 Key-Value Store Software

We use Memcached—a popular, open-source key-value store implementation—in all of our experiments. Memcached performs a few simple operations on the cache, the primary operations being *GET*, *SET*, and *DELETE*. The Memcached software itself performs little computation, and operates on relatively small in-memory objects.

We compile Memcached version 1.4.15, and libevent version 2.0.21, for the ARMv7. Libevent is a library that uses the `poll()` system call as a backend. Memcached uses libevent to perform non-blocking I/O. We modify the Memcached source to use our networking API, as opposed to the Berkeley sockets API. The changes required are minimal as we only need to modify the code responsible for the management of sockets.

4.5.2 Memcached Client

For our client we use the Memcached client that is included in *CloudSuite* [22]. The Memcached client uses a popularity distribution of object sizes to generate requests. A user can specify a desired RPS value—which is the number of requests the client will attempt to send each second—as well as the ratio of GETs to SETs. The number of client connections and threads can also be specified. The client generates requests in an open-loop fashion, meaning it does not wait for a response before sending additional requests. The client outputs several important metrics: the achieved RPS, the average request latency, and the tail latencies. The client assumes a strict SLA that requires 95% of all requests be satisfied within 10ms.

To generate requests we use two workloads: the *FriendFeed* and *MicroBlog* workloads as described in [56]. Both workloads use an object size distribution that is based on a sample of "tweets" collected from Twitter, and is available as part of CloudSuite. These workloads represent the objects that are typically cached by popular social networking sites such as Twitter and Facebook. The messages are usually brief—a max of 140 characters in the case of Twitter—leading to an average object size of approximately 1KB. The FriendFeed and MicroBlog workloads use the same object distribution, as prior work has shown that Facebook status updates and Twitter tweets have similar object sizes and popularity distributions [3]. The FriendFeed workload, however, uses MULTI-GET requests, which are central in Facebook's Memcached implementation [20]. As with the Memcached server software, we modify the client software to use our networking API in place of Berkeley sockets and compile for the ARM v7 ISA.

4.5.3 Simulation Infrastructure

To evaluate the performance of a KeyVault server, and compare it to the Mercury server, we use the *gem5* full-system simulator [10]. gem5 is capable of modeling multiple networked systems with detailed CPU, memory, and device models. It is capable of running full operating systems and networking stacks. Table 4.1 shows a detailed breakdown of the components on a single KeyVault stack. For each stack modeled in gem5, we use the *O3CPU* model in a Cortex-A15-like configuration. We implement our integrated NIC and the KVD in gem5, and all networking devices are connected with 10Gbps network links. The Mercury stack is modeled nearly identically with the exception of the on-stack NIC. KeyVault uses our custom integrated NIC design, while the Mercury stack has a standard Ethernet NIC and uses the Intel e1000 network driver. The Mercury stacks also have dedicated 10GbE links, whereas our KeyVault stacks share a single 10GbE port through the KVD.

In all of our experiments we have a single client system and one or more server systems. We fast-forward through the kernel boot process and warmup the Memcached server with functional simulation. We then switch to the detailed O3CPU model and simulate for 60 seconds, taking a sample of the sustained RPS, and average latency values every second. We sweep through a variety of attempted RPS values, from 1,000 RPS to 5 million RPS.

4.5.4 Operating System

The client and server software are run on Ubuntu server version 11.04, with version 3.3.0-rc3 of the Linux kernel, both built for the ARM v7 ISA. PCIe support is added to the kernel via the patch on the gem5 website. We used this kernel in all of our simulations and our NIC driver is built against its source.

4.5.5 Power Modeling

To calculate the total power budget for a KeyVault server we first assume a 750W power supply from HP [33]. We allocate 160W for miscellaneous components and the motherboard. We then assume a 20% margin for power and delivery losses, which is a conservative estimate. This leaves a total power budget of 472W for a 1.5U KeyVault server, including KVDs, cores, memory, etc.

We list the total power for each component of a KeyVault server in table 4.1. The total power of a KeyVault server is the sum of the power of each component in the 1.5U box: CPUS, memory, NICs, and KVDs. The on-stack NIC power estimates are drawn from the Niagara-2 design [7, 51], while the 10GbE PHY power is based on a Broadcom part [12]. To obtain power estimates for the KVD switching fabric, we implement its design in HDL and synthesize it in ST's 28nm FDSOI process. The power estimates for the buffers in all networking hardware come from CACTI [68]. The Cortex-A15 power is obtained from [30], and the 3D-stacked DRAM power is calculated from a technical specification obtained from Tezzaron [26]. The amount of power consumed by the 3D-stacked DRAM is dependent on the bandwidth utilization, therefore the power number reported in table 4.1 is per GB/s.

4.5.6 Area Estimation

The area of the on-stack NIC is obtained by scaling the Niagara-2 NIC to 28nm. For the KVDs, we synthesize an HDL model in ST's 28nm FDSOI process. The buffer area for all of the networking hardware is obtained from CACTI. The area of the Cortex-A15 CPU is taken from [30], and the area estimates for the Tezzaron Octopus DRAM were obtained from Tezzaron [26]. We can fit several hundred cores on a single stack, over 400, however we limit the number of cores on a stack to 16 because the 3D-stacked DRAM has 16 ports.

In a single 1.5U server we place as many clusters of KeyVault stacks and KVDs as we can fit into our power and area budget. Each stack is packaged in a 441mm^2 BGA package, and two PHYs are packaged into a single 441mm^2 BGA package. Due to pin counts, the KVDs must be placed into 1,225mm^2 BGA packages. If we assume 75% utilization of a 1.5U, 13in$\times$13in motherboard for KeyVault stacks and KVDs—and

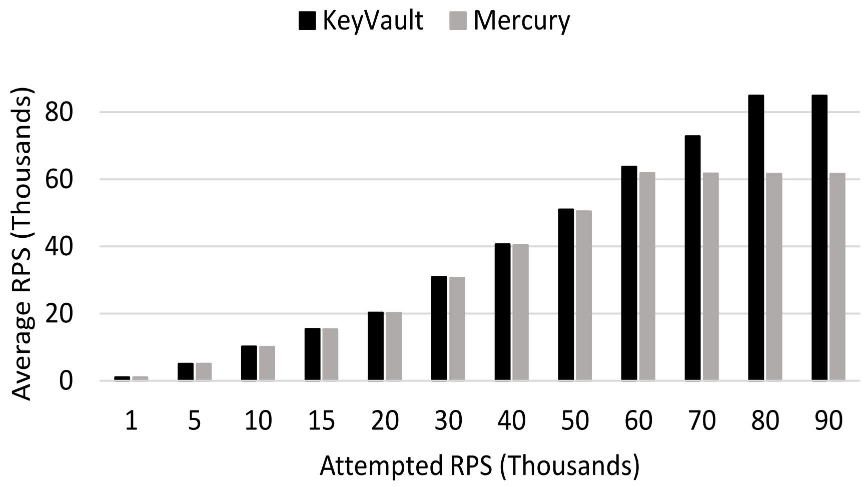


Figure 4.4: Average RPS vs. attempted RPS for the FriendFeed workload.

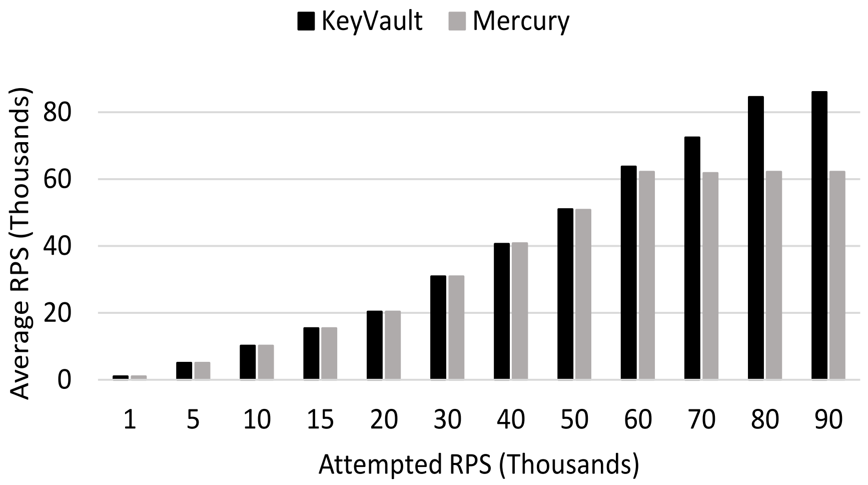


Figure 4.5: Average RPS vs. attempted RPS for the MicroBlog workload.

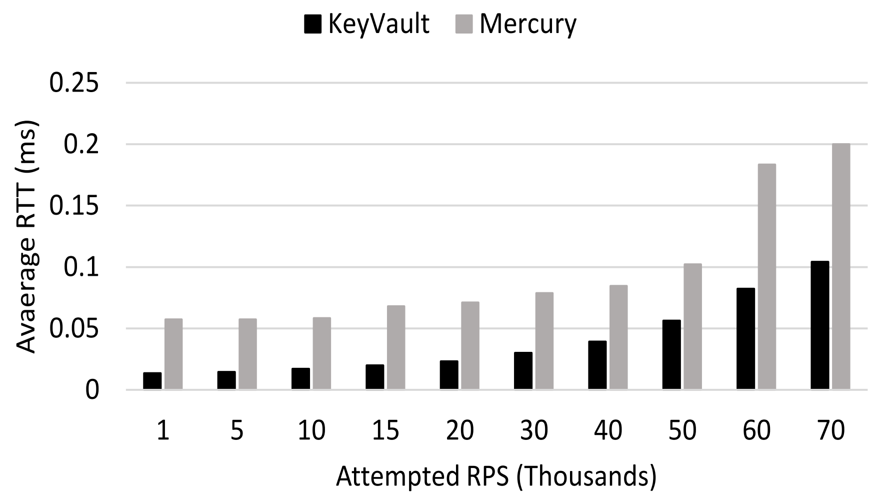


Figure 4.6: Average request latency vs. attempted RSP for the FriendFeed workload.

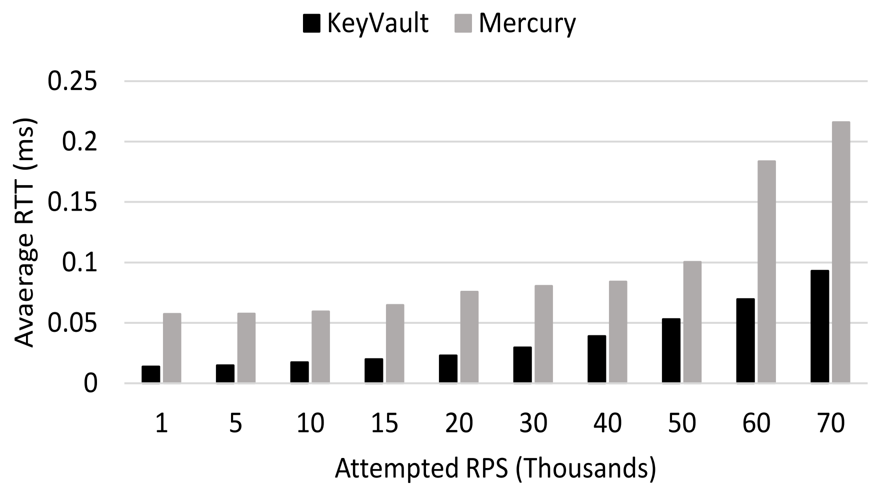


Figure 4.7: Average request latency vs. attempted RSP for the MicroBlog workload.

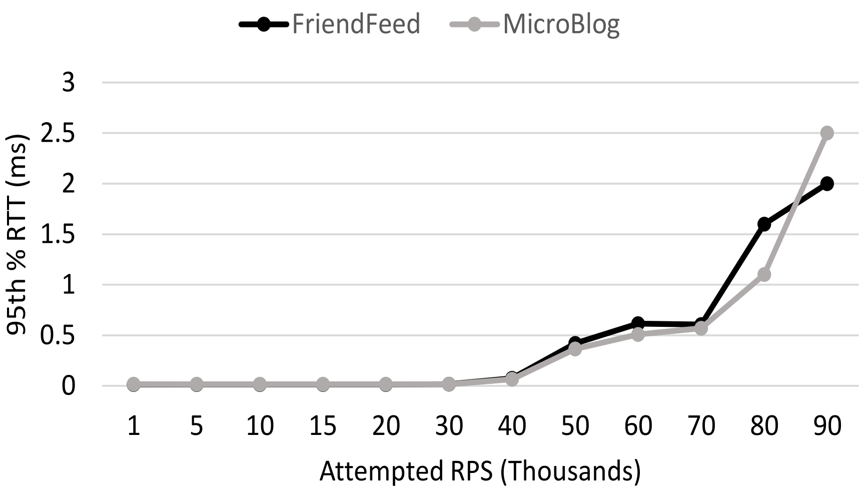


Figure 4.8: 95% Latency vs. attempted RSP for FriendFeed and MicroBlog on KeyVault.

12 stacks per KVD—then we can fit roughly 144 stacks in a single 1.5U server.

4.5.7 Cooling

The TDP of a KeyVault stack is roughly 600W, which is similar to current 1.5U systems, and is spread across all stacks. This yields a TDP of under 5W per stack. We expect that the power dissipated by each stack will be well within the capabilities of passive cooling, while a fan is used to extract heat from the 1.5U box. Prior work on thermal characterization of cloud workloads also supports this [65].

4.6 Results

In order to make a direct comparison with Mercury, we do a stack-to-stack comparison of a single KeyVault stack and a single Mercury stack, both containing a single CPU. We then compare the maximum achievable performance of a complete KeyVault server—including the KVD and clustered architecture—and a complete Mercury server.

4.6.1 Request Throughput

We first find the average sustainable request throughput for both a single KeyVault and Mercury stack containing a single CPU. In this experiment we run a sweep for a variety of attempted RPS, up to several million, however the sustainable RPS saturates at less than 100,000 RPS for both the KeyVault and Mercury stacks. Figures 4.4 and 4.5 show the average sustainable RPS for the FriendFeed and MicroBlog workloads respectively, over a variety of attempted RPS. The x-axis represents the throughput the client attempts to achieve, and the y-axis represents the actual throughput that is achieved on average. The peak average throughput that is sustainable for both the FriendFeed and MircoBlog workload is around 85,000 RPS on a KeyVault stack, when queueing becomes the limiting factor. Both workloads have a maximum average sustainable throughput of around 61,000 RPS on a Mercury stack. The highly integrated NIC—with its zero-copy capability—on the KeyVault stack allows for a throughput that is approximately 40% greater than that of a Mercury stack.

4.6.2 Request Latency

While throughput is an extremely important metric, the average latency—as well as the tail latency—of requests remains an important metric for evaluating QoS [18]. Scale-out workloads generally try to reach a target throughput while still meeting specific latency, sometimes referred to as a SLA. We compare the the average RTT for requests on a KeyVault stack against the average RTT of requests on a Mercury stack, we then report the 95th percentile latency of a KeyVault stack.

Figure 4.9 shows the amount of time, on average, spent in the network stack, hash computation, or miscellaneous Memcached code for a single request in the MicroBlog workload when attempting 50 thousand RPS. When compared to figure 3.4 we can see that the amount of time spent in the network stack decreases from over 90%—for

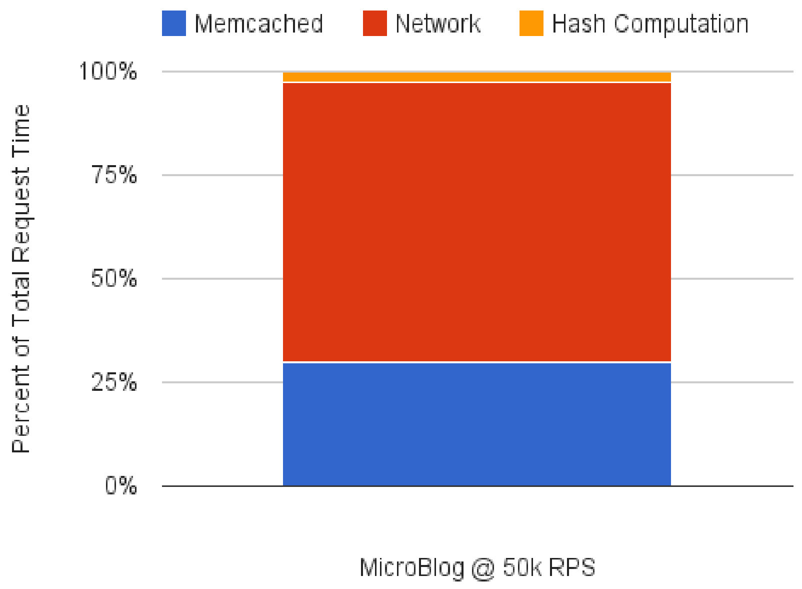


Figure 4.9: Percentage of time spent in the networking stack, performing the hash computation, or executing miscellaneous Memcached code on average for a single request.

the 1k requests, which is the average size of requests in the MicroBlog workload—to around 66%. Because the network stack dominates, this near 25% decrease in the time spent in the network stack on average, we a see a near 2X reduction in overall RTT.

4.6.2.1 Average Latency

Figures 4.6 and 4.7 show the average request latency the FriendFeed and MicroBlog workloads respectively. Because the Mercury stack peaks at an attempted RPS of approximately 61,000 these graphs only show up to an attempted 70,000 RPS. For both workloads the KeyVault stack is able to provide several times better latency, and at peak the RTT of requests on a Mercury stack is 2.3× higher than on a KeyVault stack. These results show just how much overhead is incurred due to the TCP/IP protocol stack and OS code.

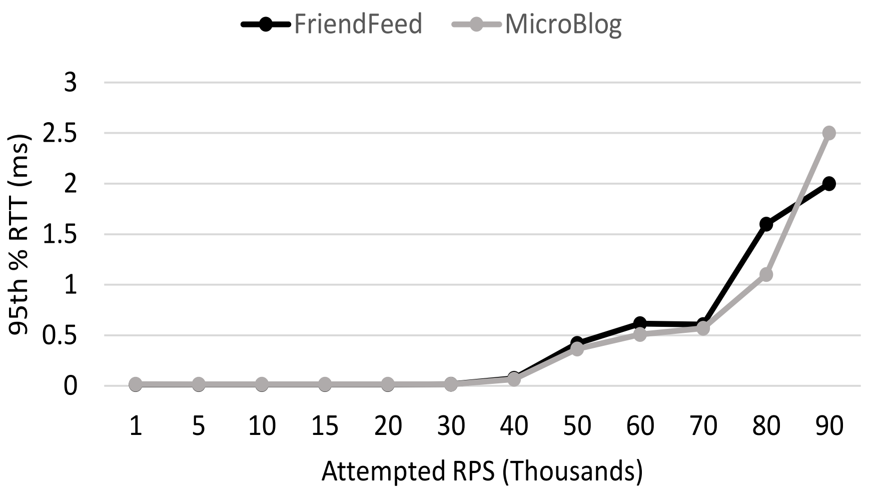


Figure 4.10: 95% Latency vs. attempted RSP for FriendFeed and MicroBlog on KeyVault.

4.6.2.2 95th Percentile Latency

The 95th percentile latency for both the FriendFeed and MicroBlog workloads is shown in figure 4.10. Both workloads exhibit very low tail latency for requests when run on a KeyVault stack. Latency increases at about 50,000 RPS when the queue length begins to increase, however even at peak throughput the 95th percentile latency for both workloads is well within the latency specified by the SLA, which is assumed to be 10ms.

4.6.3 Network Link Bandwidth

One of the key sources of inefficiency when running Memcached is network bandwidth utilization. Figures 4.11 and 4.11 show the network bandwidth in Mbps for the FriendFeed and MicroBlog workloads respectively. By reducing the per-packet overhead and providing very fast NICs with zero-copy capabilities, a KeyVault stack is able to achieve about 60% higher network bandwidth when compared to a Mercury stack. This is a substantial increase, but the primary benefit comes with the addition of the KVD. By sharing a single 10GbE link amongst several KeyVault stacks, we

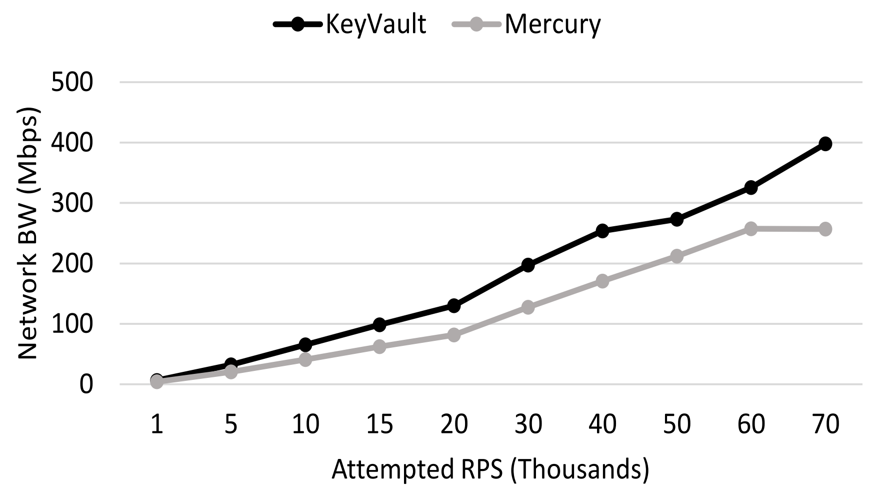
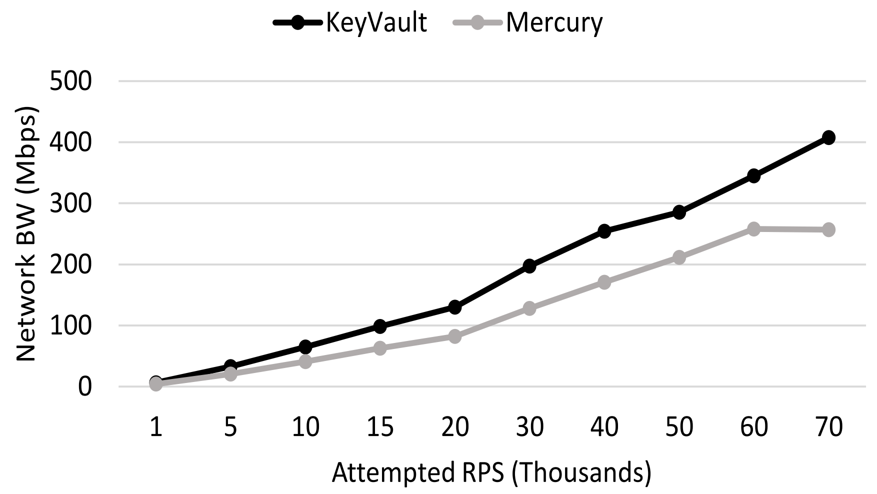


Figure 4.11: Sustained network bandwidth for the FriendFeed workload.



Sustained network bandwidth for the MicroBlog workload.

can achieve nearly 100% link utilization per link—as we will see in section 4.6.5—and reduce the number of 10GbE links required, without sacrificing throughput.

4.6.4 Memory Bandwidth

Memory bandwidth is also underutilized by Memcached. In figures 4.12 and 4.13 we report the memory bandwidth for both the FriendFeed and MicroBlog workloads respectively. The bandwidth is reported in MB/s. Because of the reduced amount of processing time spent on executing network code, more requests can be sent to the stack's memory in a KeyVault server when compared to a Mercury server. This

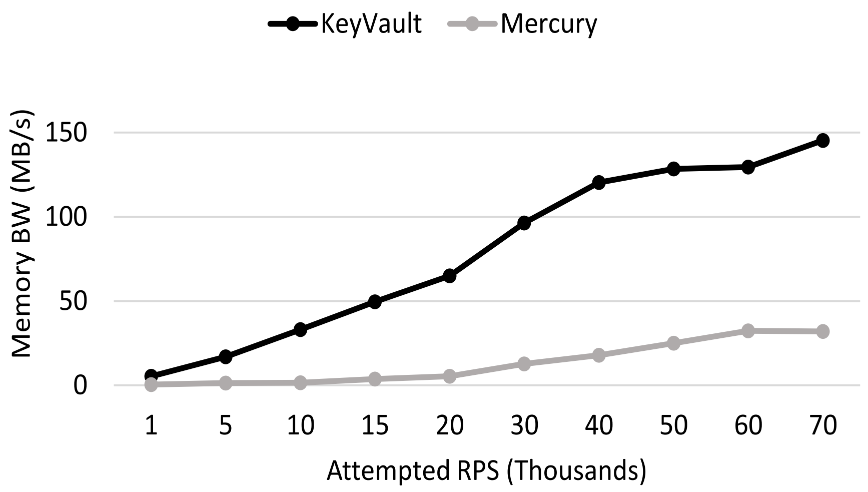


Figure 4.12: Sustained memory bandwidth for the FriendFeed workload.

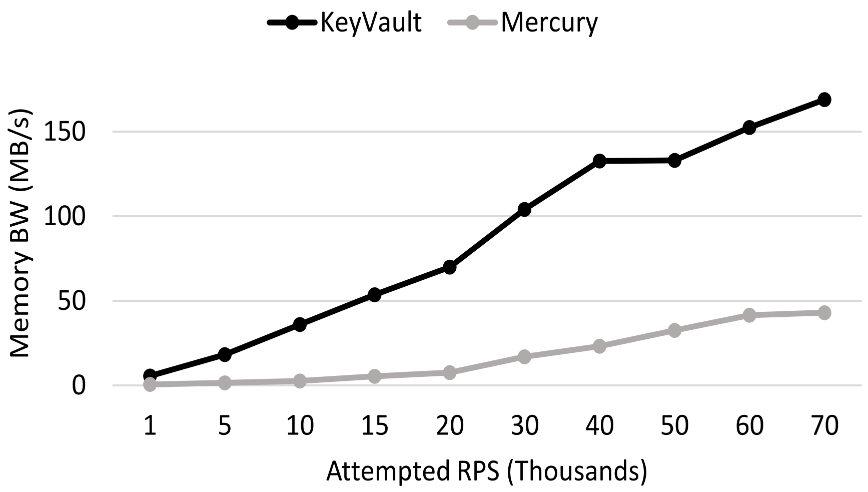


Figure 4.13: Sustained memory bandwidth for the MicroBlog workload.

leads to a 4.5× improvement in bandwidth for a KeyVault stack at peak throughput when compared to a Mercury stack. While the overall memory utilization remains low on a KeyVault stack, the large amount of bandwidth available from the 3D-stacked memory comes at a low cost. We could also explore the option of reducing the 3D-stacked memory's maximum bandwidth in order to acquire additional power savings, but that is beyond the scope of this work.

4.6.5 Maximum 1.5U Server Performance

Here we provide a design-space exploration for several 1.5U KeyVault server configurations. We contrast these with equivalent 1.5U Mercury server configurations. 4.2. We look at several different versions of each server—based on the number of cores per stack—with the goal of fitting as much compute and memory into a single server as possible, while remaining within the power budget described in section 4.5.5.

The columns of table 4.2 each represent the maximum performance we can extract from a 1.5U server box with n cores per stack, where $n \in 2, 4, 8, 16$. We can see that as we increase the number of cores per stack the total throughput delivered by the a 1.5U server increases for both the KeyVault and Mercury servers, however the smaller core counts deliver much higher memory density, while still maintaining relatively high throughput. This is primarily due to power limitations; as the core counts increase, power becomes a limiting factor and the number of stacks we can fit in our power budget decreases. For example, when we have 16 cores per stack we are limited to 25 stacks in a 1.5U KeyVault server. In this scenario it is more advantageous to have each stack directly connected its own 10GbE port, foregoing the use of a KVD. The added throughput we achieve comes at the cost of sacrificed memory density and network bandwidth utilization. The Mercury servers also sacrifice memory density as they scale up cores. Because we are primarily concerned with memory density, we choose the design point with two cores per stack as our desired configuration.

		Mercury				KeyVault			
	Version	n=2	n=4	n=8	n=16	n=2	n=4	n=8	n=16
1.5U Server	Stacks	96	87	50	27	144	90	48	25
	KVDs	—	—	—	—	12	15	16	0
	Cores	192	348	400	432	288	360	384	400
	10 GbE Ports	96	87	50	27	12	15	16	25
	Memory(GB)	384	348	200	108	576	360	192	100
	Power(W)	570	749	749	748	728	748	725	738
	RPS(millions)	11.93	21.63	24.86	26.85	24.79	30.99	33.05	34.43
	RPS(thousands)/Watt	20.92	28.86	33.20	35.90	34.01	41.45	45.62	46.66
	RPS(thousands)/GB	31.08	62.15	124.31	248.61	43.04	86.07	172.14	344.39
	Memory BW Utilization	0.08%	0.17%	0.34%	0.67%	3.96%	3.96%	3.96%	2.64%
	Network BW Utilization	5%	11%	20%	40%	96%	96%	96%	64%

Table 4.2: Comparison of various KeyVault and Mercury server configurations. A single KeyVault server is able to provide 2× the throughput of a Mercury server. Memory density is improved over a Mercury server by 50%. This improved performance comes while simultaneously improving power and bandwidth efficiency.

	Mercury	KeyVault	Commodity	MICA	HERD	TSSP
RPS	12 million	25 million	3.15 million	8.6 million	5 million	0.3 million
RPS/W	21 thousand	34 thousand	11 thousand		3.6 thousand	17.63 thousand
RTT	3µs	90µs	300µs	52µs	5µs	20µs

Table 4.3: Comparison of KeyVault to several prior works.

A 1.5U KeyVault server with two cores per stack has a 50% larger memory capacity, and over 2× higher sustained throughput compared to Mercury. Power efficiency is also improved as a single 1.5U KeyVault server has 63% higher RPS/W than a Mercury server. Memory bandwidth is approximately 45× higher, and network link bandwidth improves from approximately 5% to nearly 100%.

4.6.6 Comparison to Prior Works

In figure 4.14 we compare the total throughput available in a single KeyVault server—for several different configurations of cores per stack—against several priors works. HERD [39] is the most recent, and highest performing, RDMA-based solution; *Thin Servers with Smart Pipes (TSSP)* [56] is a hardware key-value store accelerator;

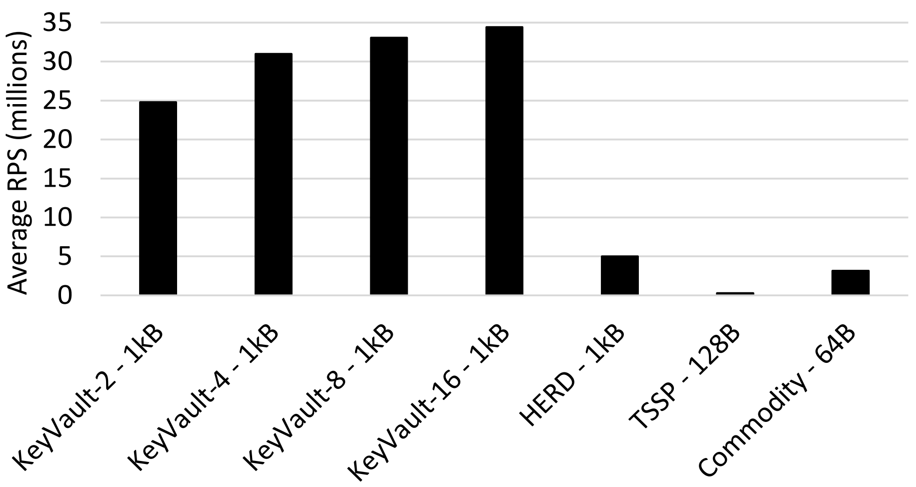


Figure 4.14: Comparison KeyVault to prior works.

and the the commodity server is based on the work in [86], where the authors run an enhanced version of Memcached on a commodity server design. Our workload uses a popularity distribution with requests that have an average value of approximately 1kB. HERD, TSSP, and the commodity server's throughput numbers, however, are based on fixed-size requests of 1kB, 128B, and 64B respectively.

By providing integrated, lightweight networking, along with servers designed for density, we can provide approximately 5× the throughput of HERD, 8× that of a commodity server, and nearly two orders of magnitude more that TSSP. The average RTT of requests on a KeyVault server is around 90µs. The average RTT is approximately 5µs, 20µs, and 300µs on HERD, TSSP, and the commodity server respectively. The HERD work only reports the latency number for 48B fixed-size requests. Additionally, TSSP's RTT does not take into account the effects of queueing. While it is difficult to directly compare throughput and latency with drastically different request sizes, it is likely that the latency would increase for the other solutions if larger request sizes are used.

4.7 Related Work

In order to provide a comprehensive rack-scale solution for efficient data centers, KeyVault builds on many concepts presented in prior works. Here we detail the pertinent works and how they relate to KeyVault.

4.7.1 Improving Key-Value Stores

Lim et al. design a key-value accelerator for use in an alternate architecture for Memcached [56]. TSSP builds on prior work that puts a similar Memcached accelerator on an FPGA fabric [14]. In TSSP a hardware GET accelerator interacts with the NIC, and essentially acts as a UDP offload engine. Because GETs make up the vast majority of requests for most workloads—and to avoid synchronization issues—only GETs are accelerated. The NIC inspects packets and if a GET request is found, the request is offloaded to the GET accelerator. The GET accelerator directly accesses the software-owned cache, without any software involvement. The GET accelerator owns the in-memory hash table; it performs a hash on the key and if found, fetches the data and builds a reply packet. KeyVault differs from TSSP by providing both integrated, intelligent network interfaces, as well as a simplified networking protocol. By simplifying the network protocol we allow the software to perform all request processing, which simplifies synchronization and improves the performance of all requests, as opposed to only GETs.

Andersen et al. propose a cluster architecture, called *FAWN*, for efficient key-value stores [2]. In their work they use a large number of ”wimpy” cores along with FLASH memory to improve the density and energy efficiency of key-value store applications. Their work primarily focuses on enabling FLASH memory, i.e., file system design. They design a log structured file system that mitigates FLASH wear out. KeyVault focuses on using 3D-stacked memory to enhance density and energy efficiency, while improving performance via highly integrated networking.

MICA [53] proposes a holistically designed key-value store. MICA is designed to overcome the bottlenecks present in typical key-value store deployments by focusing on the following design goals: fast parallel data access, low overhead networking software, and optimized data structures. The most relevant portion of the MICA work with respect to this dissertation is the low-overhead networking stack. MICA, however, utilizes direct NIC access from userspace software, which limits the size of the requests it can service. This is ok for MICA's design goals, which aim to optimize for very small key-value pairs. However, for more general key-value deployments this solution is not ideal.

Pilaf [66] and Herd [39] propose using RDMA to reduce the networking overhead for key-value requests. Pilaf uses single-sided RDMA READs to completely bypass the CPU when processing GET requests. Because they are completely bypassing the CPU each request requires multiple trips across network links and require a sophisticated synchronization scheme. HERD, on the other hand, uses RDMA writes to put a request's payload directly into memory, which is similar to our design. However, our design provides for tighter integration, which allows for greater throughput and network density.

4.7.2 Remote Memory Access

Scale-Out NUMA (soNUMA) [70] is a recent work that aims to reduce the overhead incurred by performing remote memory accesses in rack-scale systems. soNUMA provides an RDMA-like programming model, and a message passing communication protocol that relies on a *Remote Memory Controller (RMC)*—a memory controller that is integrated directly into the cache hierarchy. soNUMA is designed for rack-scale workloads, and is built on top of a NUMA fabric, because of this it requires that its system provide the appearance of a global address space. To simplify their protocol and hardware design, they only transfer objects a single cache line at a time. For rack-scale deployments they are able to achieve remote memory reads that are within a few times that of local reads. With KeyVault servers, we target internet services in particular—where many processes are attempting to access memory remotely, and a single global address space is difficult to achieve. We imagine a deployment where accesses can come from many different networks. Our networking protocol is also able to transmit much larger data objects—packets are encapsulated in standard Ethernet packets at the data link layer—which is more suitable for internet services.

4.7.3 Integrated Networking

Several works have investigated reducing the network overhead by providing integrated networking hardware. Mukherjee et al. [67] proposed one of the earliest examples of integrating networking with the cache hierarchy. They propose *coherent network interfaces (CNI)*, which utilize a cacheable device register—a cacheable region shared by the NIC and the CPU—to transfer status, control or data information quickly. *Direct Cache Access (DCA)* [35] provides a method that allows the I/O controller to directly insert packet information into a processor's cache, thereby reducing latency, improving throughput, and lowering cache miss rates. *SINIC* and *V-SINIC* [11] simplify the NIC and integrate it closely with the CPU, while work that is traditionally done by the NIC is pushed into the software. This allows for optimizations that are better suited for software, as well as true zero-copy receives. All of these works are done in the context of the TCP/IP stack. With KeyVault servers we advocate rethinking the entire networking stack to enable higher performance and throughput.

4.7.4 Scale-Out Server Design

Several prior studies have explored using 3D-stacked memory as a means to improve density and energy efficiency. PicoServer [42] utilizes 3D stacking technology to design compact and efficient multi-core processors for tier 1 internet services. Their primary focus is on energy efficiency. By introducing 3D-stacked memory into their design they remove the need for complex cache hierarchies, and add more CPUs, thereby improving throughput while saving energy. Nanostores [15] builds on the PicoServer design by introducing FLASH or Memristor memory technology into the stack. NDC [75] explores the use of 3D-stacked memory with MapReduce workloads. While these works focus primarily on improving throughput and energy efficiency, our work recognizes the trend of internet services operating on massive, distributed data sets and considers density a first-class design constraint. We try to fit as much compute and memory as possible into a single KeyVault server, while providing fast access via highly integrated networking.

More recently, Scale-Out Processors [59] were proposed as a solution for cloud computing workloads. They propose clustering groups of cores into *pods*—each with their own coherence domain—and connecting each pod together using a fast interconnect. Our KeyVault servers use a pod-like organization as well, but supplement the server design with 3D-stacked DRAM and highly integrated networking to further improve density and throughput.

3D-stacked memory has also been produced for use as a cache. Footprint Cache [37] uses 3D-stacked DRAM caches along with a clever prefetching scheme to improve performance and reduce energy. In KeyVault servers we propose using 3D-stacked DRAM as the primary source for memory, while focusing on compute and memory density.

4.8 Conclusion

In this work we present KeyVault, a server architecture that aims to provide a complete solution for modern data center workloads. By integrating 3D-stacked DRAM, low-power CPUs, and networking hardware—along with a simplified networking protocol—we are able to achieve a throughput of approximately 25 million RPS in a single 1.5U enclosure at a latency of less than 100µs. As the amount of data in the cloud grows, commodity servers can no longer scale, and are not a viable option going forward. We believe that a holistic approach that specializes hardware and software together will enable further scalability, and have explored this with our KeyVault design.

CHAPTER V

Conclusion

This dissertation makes a case for the use of physically dense servers. It presents the physical design of a dense server in chapter III. I detailed design space exploration is provided showing the benefits of such designs for big data workloads. The primary benefits being improved memory density and power efficiency, while maintaining QoS guarantees. However, physical density is shown to not be enough because the network bottleneck remains. Big data workloads—and key-value stores in particular—service many requests to small disparate pieces of data.

In order to overcome the networking bottlenecks present for key-value stores, chapter IV presents the design of physically dense servers with integrated networking. By including integrated NICs and an integrated switching fabric, a lightweight networking stack may be used. The insight is that—unlike prior approaches on accelerating key-value stores, which try to process requests entirely without CPU involvement—zero-copy packet placement is sufficient to remove the networking bottleneck.

Going forward, the amount of data in the cloud will grow, as will the number of users making requests on that data. Commodity off-the-shelf hardware has been shown to be inefficient when running modern cloud services. In order to scale the cloud, and to ensure its operation remains cost-effective, alternative technologies should be utilized.

5.1 Future Research Directions

This work has explored the design of physically dense servers using state-of-the-art 3D-stacked memories, in particular DRAM and Flash. There are several other types of memory technologies, such as Phase-change RAM, that may provide further benefits in terms of density, performance, and efficiency. Exploring these various memory technologies to determine which is best-suited for physically dense server design is one line of further research we may wish to explore.

In this work we have assumed a simple system-level design for our 3D-stacks—the CPUs are not kept coherent and contain only private, per-core caches. While this approach provides simplicity, and is effective for key-value stores, there may be more appropriate designs for more general server applications. Exploring alternative system-level designs is something we may also explore.

One final aspect of physically dense server design that will be explored is the design of system-level software. Given our unique server design, and use of NVRAM as primary memory, we may explore file-system and operating system software that takes into account both density and the reliability of NVRAM cells.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Louis Abate. Comparing Cloud Storage Providers in 2013—Infographic. `http://www.nasuni.com/193-comparing_cloud_storage_providers_in/`, 2013.

[2] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *the proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 1–14, 2009.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *the proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[5] Yong-Cheol Bae, Joon-Young Park, Sang Jae Rhee, Seung Bum Ko, Yonggwon Jeong, Kwang-Sook Noh, Younghoon Son, Jaeyoun Youn, Yonggyu Chu, Hyunyoon Cho, Mijo Kim, Daesik Yim, Hyo-Chang Kim, Sang-Hoon Jung, Hye-In Choi, Sungmin Yim, Jung-Bae Lee, Joo-Sun Choi, and Kyungseok Oh. A 1.2V 30nm 1.6Gb/s/pin 4Gb LPDDR3 SDRAM with Input Skew Calibration and Enhanced Control Scheme. In *proceedings of the 2012 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 44–46, 2012.

[6] Jeff Barr. Amazon S3—The First Trillion Objects. `https://aws.amazon.com/blogs/aws/amazon-s3-the-first-trillion-objects/`, 2012.

[7] Jama Barreh, Jeff Brooks, Robert Golla, Greg Grohoski, Rick Hetherington, Paul Jordan, Mark Luttrell, Chris Olson, and Manish Shah. Niagara-2: A Highly Threaded Server-on-a-Chip. In *the proceedings of the 18th Hot Chips Symposium*, 2006.

[8] Luiz Andr Barroso and Urs Hlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines (2nd ed.)*. 2013.

[9] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-Core Key-Value Store. In *proceedings of the 2011 International Green Computing Conference and Workshops (IGCC)*, pages 1–8, 2011.

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *SIGARCH Computer Architer News*, 39(2):1–7, 2011.

[11] Nathan L. Binkert, Ali G. Saidi, and Steven K. Reinhardt. Integrated Network Interfaces for High-Bandwidth TCP/IP. In *the proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 315–324, 2006.

[12] Broadcom. Broadcom Introduces Industry's Lowest Power Octal 10GbE PHY. `http://www.broadcom.com/press/release.php?id=s706156`, 2012.

[13] V. Cerf and R.E. Kahn. A Protocol for Packet Network Intercommunication. *Communications, IEEE Transactions on*, 22(5):637–648, 1974.

[14] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. An FPGA Memcached Appliance. In *the proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 245–254, 2013.

[15] Jichuan Chang, Parthasarathy Ranganathan, Trevor Mudge, David Roberts, Mehul A. Shah, and Kevin T. Lim. A Limits Study of Benefits from Nanostore-Based Future Data-Centric System Architectures. In *proceedings of the 9th Conference on Computing Frontiers (CF)*, pages 33–42, 2012.

[16] J. Clidaras, D.W. Stiver, and W. Hamburgen. Water-Based Data Center. `https://www.google.com/patents/US7525207`, April 2009.

[17] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW Client-based Traces. Technical Report TR-95-010, Boston University, 1995.

[18] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[19] Ashutosh Dhodapkar, Gary Lauterbach, Sean Lie, Dhiraj Mallick, Jim Bauman, Sundar Kanthadai, Toru Kuzuhara, Gene Shen, Min Xu, and Chris Zhang. Building Datacenter Servers Using Cell Phone Chips. In *the proceedings of the 23rd Hot Chips Symposium*, 2011.

[20] Facebook. Facebook and Memcached with Mark Zuckerberg. `https://www.youtube.com/watch?v=UH7wkvcf0ys`, 2008.

[21] W.M. Felter, T.W. Keller, M.D. Kistler, C. Lefurgy, K. Rajamani, R. Rajamony, F.L. Rawson, B. A. Smith, and E. Van Hensbergen. On the Performance and Use of Dense Servers. *IBM Journal of Research and Development*, 47(5.6):671–688, 2003.

[22] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *the proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, 2012.

[23] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Quantifying the Mismatch Between Emerging Scale-Out Applications and Modern Processors. *ACM Transactions on Computing Systems*, 30(4):15:1–15:24, 2012.

[24] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Quantifying the Mismatch Between Emerging Scale-Out Applications and Modern Processors. *ACM Transactions on Computing Systems*, 30(4):15:1–15:24, 2012.

[25] Alex Gartrell, Mohan Srinivasan, Bryan Alger, and Kumar Sundararajan. McDipper: A Key-Value Cache for Flash Storage. `http://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-for-flash-storage/10151347090423920`, 2013.

[26] Bharan Giridhar, Michael Cieslak, Deepankar Duggal, Ronald Dreslinski, Hsing Min Chen, Robert Patti, Betina Hold, Chaitali Chakrabarti, Trevor Mudge, and David Blaauw. Exploring DRAM Organizations for Energy-efficient and Resilient Exascale Memories. In *the proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 23:1–23:12, 2013.

[27] Bharan Giridhar, Michael Cieslak, Deepankar Duggal, Ronald G. Dreslinski, Robert Patti, Betina Hold, Chaitali Chakrabarti, Trevor Mudge, and David Blaauw. Exploring DRAM Organizations for Engery-Efficient and Resilient Exascale Memories. In *proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 23:1–23:12, 2013.

[28] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steve Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing Flash Memory: Anomalies, Obervations, and Applications. In *proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 24–33, 2009.

[29] Anthony Gutierrez, Michael Cieslak, Bharan Giridhar, Ronald G. Dreslinski, Luis Ceze, and Trevor Mudge. Integrated 3D-stacked Server Designs for Increasing Physical Density of Key-value Stores. In *the proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 485–498, 2014.

[30] Linley Gwennap. How Cortex-A15 Measures Up. *Microprocessor Report*, May 2013.

[31] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[32] HP. *HP Common Slot Platinum Power Supplies*, 2010.

[33] HP. HP Common Slot Platinum Power Supplies, 2010.

[34] HP. HP Moonshot System: The world's first software defined servers, 2012.

[35] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *the proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 50–59, 2005.

[36] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[37] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *the proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 404–415, 2013.

[38] Jose Jithin, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasiur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *the proceedings of the 2011 International Conference on Parallel Processing (ICPP)*, pages 743–752, 2011.

[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *the proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 295–306, 2014.

[40] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663, 1997.

[41] R. Katsumata, M. Kito, Y. Fukuzumi, M. Kido, H. Tanaka, Y. Komori, M. Ishiduki, J. Matsunami, T. Fujiwara, Y. Nagata, Li Zhang, Y. Iwata, R. Kirisawa, H. Aochi, and A. Nitayama. Pipe-Shaped BiCS Flash Memory with 16 Stacked Layers and Multi-Level-Cell Operation for Ultra High Density Storage Devices. In *proceedings of the 2009 Symposium on VLSI Technology*, pages 136–137, 2009.

[42] Taeho Kgil, Shaun D'Souza, Ali Saidi, Nathan Binkert, Ronald Dreslinski, Trevor Mudge, Steven Reinhardt, and Krisztian Flautner. PicoServer: Using 3D Stacking Technology to Enable a Compact Energy Efficient Chip Multiprocessor. In *the proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, 2006.

[43] Taeho Kgil and Trevor Mudge. FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers. In *the proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 103–112, 2006.

[44] Taeho Kgil and Trevor Mudge. FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers. In *proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 103–112, 2006.

[45] Taeho Kgil, David Roberts, and Trevor Mudge. Improving NAND Flash Based Disk Caches. In *the proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, pages 327–338, 2008.

[46] Taeho Kgil, David Roberts, and Trevor Mudge. Improving NAND Flash Based Disk Caches. In *proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, pages 327–338, 2008.

[47] Jung-Sik Kim, Chi Sung Oh, Hocheol Lee, Donghyuk Lee, Hyong-Ryol Hwang, Sooman Hwang, Byongwook Na, Joungwook Moon, Jin-Guk Kim, Hanna Park, Jang-Woo Ryu, Kiwon Park, Sang-Kyu Kang, So-Young Kim, Hoyoung Kim, Jong-Min Bang, Hyunyoon Cho, Minsoo Jang, Cheolmin Han, Jung-Bae Lee, Kyehyun Kyung, Joo-Sun Choi, and Young-Hyun Jun. A 1.2V 12.8GB/s 2Gb Mobile Wide-I/O DRAM with 4x128 I/Os Using TSV-Based Stacking. In *proceedings of the 2011 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 496–498, 2011.

[48] P.M. Kogge. EXECUBE-A New Architecture for Scaleable MPPs. In *the proceedings of the 1994 International Conference on Parallel Processing (ICPP) Vol. 1*, pages 77–84, 1994.

[49] P.M. Kogge, S.C. Bass, J.B. Brockman, D.Z. Chen, and E. Sha. Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies. In

the proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computing, pages 88–97, 1996.

[50] P.M. Kogge, T. Sunaga, H. Miyataka, K. Kitamura, and E. Retter. Combined DRAM and Logic Chip for Massively Parallel Systems. In *the proceedings of the Sixteenth Conference on Advanced Research in VLSI*, pages 4–16, 1995.

[51] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[52] Harry Li and Amir Michael. *Intel Motherboard Hardware v1.0.* Open Compute Project, 2013.

[53] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *the proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[54] K. Lim, D. Meisner, A. Saidi, P. Ranganathan, and T. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *the proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[55] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *the proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278, 2009.

[56] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *the proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 36–47, 2013.

[57] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, pages 315–326, 2008.

[58] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *the proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[59] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. Scale-Out Processors. In *proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 500–511, 2012.

[60] David Meisner, Brian T. Gold, and Thomas F. Wenisch. PowerNap: Eliminating Server Idle Power. In *the proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–216, 2009.

[61] Mellanox Technologies. *SX1016 64-Port 10GbE SDN Switch System*, 2013.

[62] Memcached. `http://www.memcached.org`.

[63] Memcached. Binary Protocol Revamped. `https://code.google.com/p/memcached/wiki/BinaryProtocolRevamped`.

[64] Cade Metz. Facebook Data Center Empire to Conquer Heartland in 2014. `http://www.wired.com/wiredenterprise/2013/04/facebook-iowa-2014/`, 2013.

[65] D. Milojevic, S. Idgunji, D. Jevdjic, E. Ozer, P. Lotfi-Kamran, A. Panteli, A. Prodromou, C. Nicopoulos, D. Hardy, B. Falsari, and Y. Sazeides. Thermal Characterization of Cloud Workloads on a Power-Efficient Server-on-Chip. In *the proceedings of the IEEE 30th International Conference on Computer Design (ICCD)*, pages 175–182, 2012.

[66] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *the proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.

[67] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-grain Communication. In *the proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, 1996.

[68] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 3–14, 2007.

[69] Nasuni. The State of Cloud Storage, 2013.

[70] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-Out NUMA. In *the proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–18, 2014.

[71] Greg Orzell and Justin Becker. Auto Scaling in the Amazon Cloud. `http://techblog.netflix.com/2012/01/auto-scaling-in-amazon-cloud.html`, 2012.

[72] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Operating System Review*, 43(4):92–105, 2010.

[73] J.T. Pawlowski. Hybrid Memory Cube: Breakthrough DRAM Performance with a Fundamentally Re-Architected DRAM Subsystem. In *the proceedings of the 23rd Hot Chips Symposium*, 2011.

[74] Associated Press. Google to invest $300m in data center in Belgium. `http://finance.yahoo.com/news/google-invest-390m-data-center-081818916.html`, 2013.

[75] Seth H. Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads. In *the proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 190–200, 2014.

[76] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. In *proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 314–325, 2010.

[77] David Roberts, Taeho Kgil, and Trevor Mudge. Using Non-volatile Memory to Save Energy in Servers. In *the proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 743–748, 2009.

[78] David Roberts, Taeho Kgil, and Trevor Mudge. Using Non-Volatile Memory to Save Energy in Servers. In *proceedings of the 2009 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 743–748, 2009.

[79] Steve Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John Ousterhout. In *the proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2011.

[80] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *the proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, 2001.

[81] Tezzaron. `http://www.tezzaron.com`.

[82] Tezzaron. Octopus 8-Port DRAM for Die-Stack Applications. `http://www.tezzaron.com/memory/datasheets/TSC10080x_0_1.pdf`, 2012.

[83] Pamela Vagata and Kevin Wilfong. Scaling the Facebook data warehouse to 300 PB. `https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/`, 2014.

[84] M.M. Waldrop. Data Center in a Box. *Scientific American*, 297(2):90–93, 2007.

[85] Greg Whalin. Memcached Java Client. `http://github.com/gwhalin/Memcached-Java-Client`.

[86] Alex Wiggins and Jimmy Langston. Enhancing the Scalability of Memcached. Technical report, Intel, 2012.

[87] Peter J. Winzer. Beyond 100G Ethernet. *IEEE Communications Magazine*, 48(7):26–30, 2010.

[88] Wireshark. `http://wireshark.org`.

[89] Mark Zuckerberg. Facebook and memcached. `http://www.facebook.com/video/video.php?v=631826881803`, 2008.