Designing New Memory Systems for Next-Generation Data Centers

Howard Mao
Randy H. Katz, Ed.
Krste Asanovi, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-240
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-240.html

December 1, 2022

Acknowledgement

I would like to thank my adviser, Prof. Randy Katz, for his mentorship and for the financial support he provided me in the last year of my PhD.

I would like to thank Prof. Krste Asanovic for financially supporting me for most of my PhD and for giving me his feedback as my second reader, and I would like to thank my third reader, Prof. Ian Holmes, for his feedback.

I would also like to thank my fellow graduate students in the ADEPT Lab: Alon Amid, David Biancolin, Adam Izraelevitz, Sagar Karandikar, Jack Koenig, Albert Magyar, Albert Ou, Nathan Pemberton, and Jerry Zhao. Together, we worked on the tools that made this work possible.

And finally, I would like to thank my wonderful parents, Cungui Mao and Ruijuan Xu, for their love and support throughout my life.

Designing New Memory Systems for Next-Generation Data Centers

by

Howard Mao

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Randy H. Katz, Chair
Professor Krste Asanović
Professor Ian Holmes

Fall 2020

Designing New Memory Systems for Next-Generation Data Centers

Abstract

Designing New Memory Systems for Next-Generation Data Centers

by

Howard Mao

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Randy H. Katz, Chair

In recent years, there has been a trend towards greater use of DRAM in data center applications. In-memory key-value stores are being used to cache or replace disk-based databases, and memory-based big data frameworks are supplanting earlier disk-based frameworks. However, process technology improvements for DRAM have not kept pace and instead have stagnated in terms of cost and density. With next-generation memory technologies like STT-MRAM, PCRAM, and RRAM still far from commercial viability, improving memory utilization is the most potentially fruitful path towards reducing the cost of data center memory in the near term. A typical way to improve utilization of a resource in a data center is to disaggregate said resource, allowing it to be shared across multiple nodes. Disaggregating memory has generally been quite difficult because the latency of a round-trip across a typical data center network is much greater than the latency of a DRAM access. However, recent work on photonic interconnects promises to deliver data center networks with much lower latencies, making the concept of data center remote memory more feasible.

In this work, we present the design of a DRAM caching remote memory system which divides a data center rack into compute-specialized and memory-specialized nodes. Each memory blade contains a fixed-function hardware controller that serves data from its large pool of DRAM to the compute blades through the rack network. Each compute blade contains a small local DRAM that is used as a cache for remote memory. This local DRAM is managed by a hardware controller which automatically refills the cache on misses by sending requests to remote memory. This system provides a global pool of memory that can be dynamically allocated among the compute blades and is transparent to software. We evaluated our system using microbenchmarks and realistic data center applications in cloud FPGA-based RTL simulations. Through these evaluations, we found that our DRAM caching system can serve data at lower latencies than earlier virtual memory-based remote memory systems and, with the aid of prefetching, can achieve performance comparable to local DRAM.

Contents

List of Figures

List of Tables

Acknowledgments

This work would not be possible without the help and support of many people, including my professors, colleagues, and family.

I would first and foremost like to thank my adviser, Prof. Randy Katz, for his mentorship and guidance over the last several years. I also thank him for the financial support he has provided me in the last year of my PhD.

I would like to thank Prof. Krste Asanovic for financially supporting me over the majority of my PhD and for giving me his feedback as my second reader, and I would like to thank my third reader, Prof. Ian Holmes, for his feedback.

I would also like to thank my fellow graduate students in the ADEPT Lab. Together, we collaborated on the various tools that made this work possible. Alon Amid, David Biancolin, Adam Izraelevitz, Sagar Karandikar, Jack Koenig, Albert Magyar, Albert Ou, Nathan Pemberton, and Jerry Zhao, thank you all for your help and camaraderie. It has truly been a pleasure. Special thanks to Jerry for allowing me to use some of his block diagrams in Chapter 4.

And finally, I would like to thank my wonderful parents, Cungui Mao and Ruijuan Xu, for their love and support throughout my life. Thank you, Mom and Dad. I couldn't have gotten here without you.

Chapter 1

Introduction

A major trend in data center workloads in the past decade has been increasing demand for memory. To deliver higher throughput or lower latency, application developers have increasingly been moving data that once resided in secondary storage into main memory. For instance, it has become common for web applications to cache the results of database queries in in-memory key-value stores like MemCached [9] or Redis [34]. This trend can also be seen in big data applications, where map-reduce frameworks which store intermediate results on disk, such as Hadoop, have been supplanted by frameworks which store intermediate results in memory, such as Apache Spark [42].

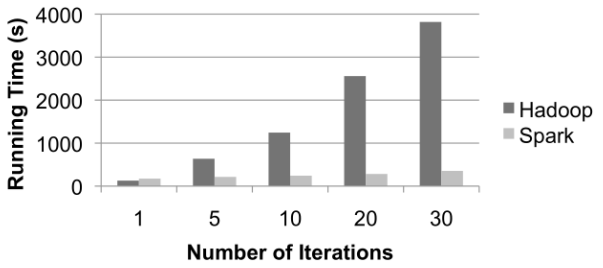


Figure 1.1: Comparison of Spark vs. Hadoop performance on Logistic Regression Benchmark [42]

1.1 The End of DRAM Cost/Density Scaling

Advancements in memory technology have not kept pace with this increasing demand. Improvements in DRAM cost have plateaued in recent years, as shown in Figure 1.2 [29]. The same can be said for memory density, as the amount of memory that can fit on a single DIMM has been stagnant at around 8GB - 16GB for several years now (Fig. 1.3).

One promising avenue for further improvements is in new non-volatile random access memory (NVRAM) technologies, such as spin-transfer torque magnetic RAM (STT-MRAM), phase-change RAM (PCRAM), and resistive RAM (RRAM). These technologies promise to deliver storage density and persistence similar to that of solid-state drives, but with a much lower latency than conventional flash memory. As shown in Figure 1.4, all three types of NVRAM have similar read latency to DRAM. However, the latency of commercially-available NVRAM is still several orders of magnitude slower than DRAM. For instance, Intel's Optane NVRAM technology boasts a read latency of 7 μs and write latency of 18 μs. As high-end DDR4 memory can offer read latencies as low as 14 ns, it is clear that there is still a long way to go before NVRAM can function as a drop-in replacement for DRAM.

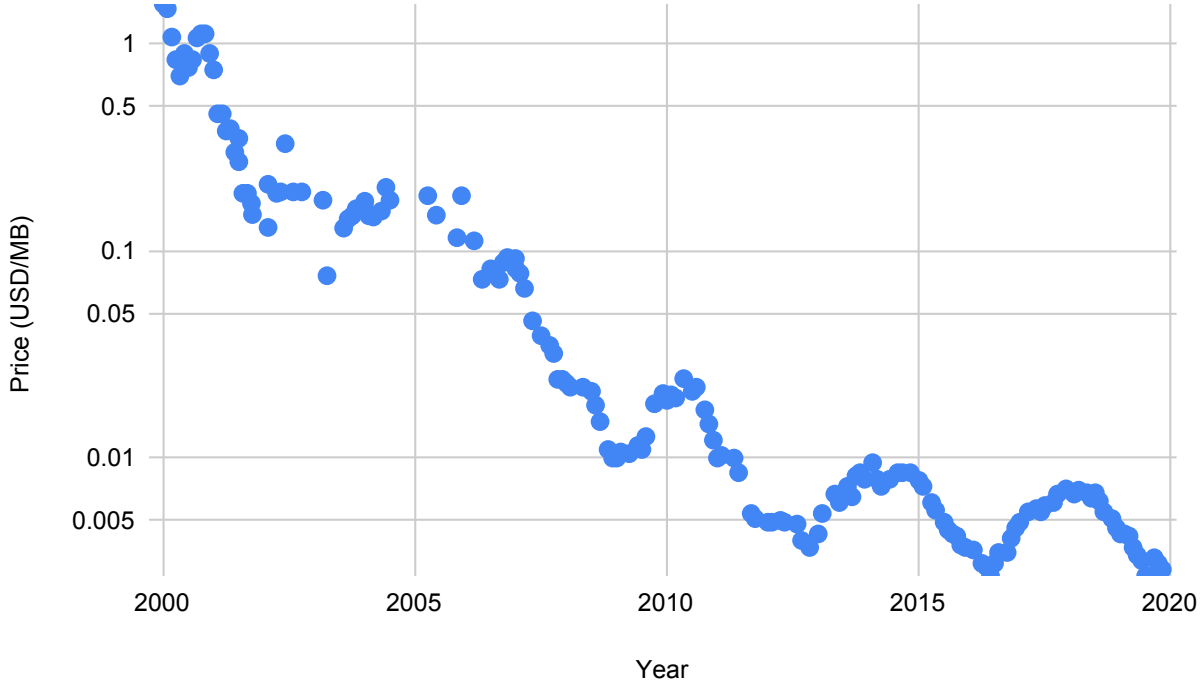


Figure 1.2: DRAM Cost over Time

1.2 Poor Utilization of Memory in Data Centers

Since cost-per-bit of conventional DRAM is no longer decreasing and new NVRAM technologies are not yet performant enough, the only way to decrease memory costs in data centers is to use the existing memory more efficiently. The status quo leaves much room for improvement, as data center memory is notoriously underutilized. Trace analysis of a

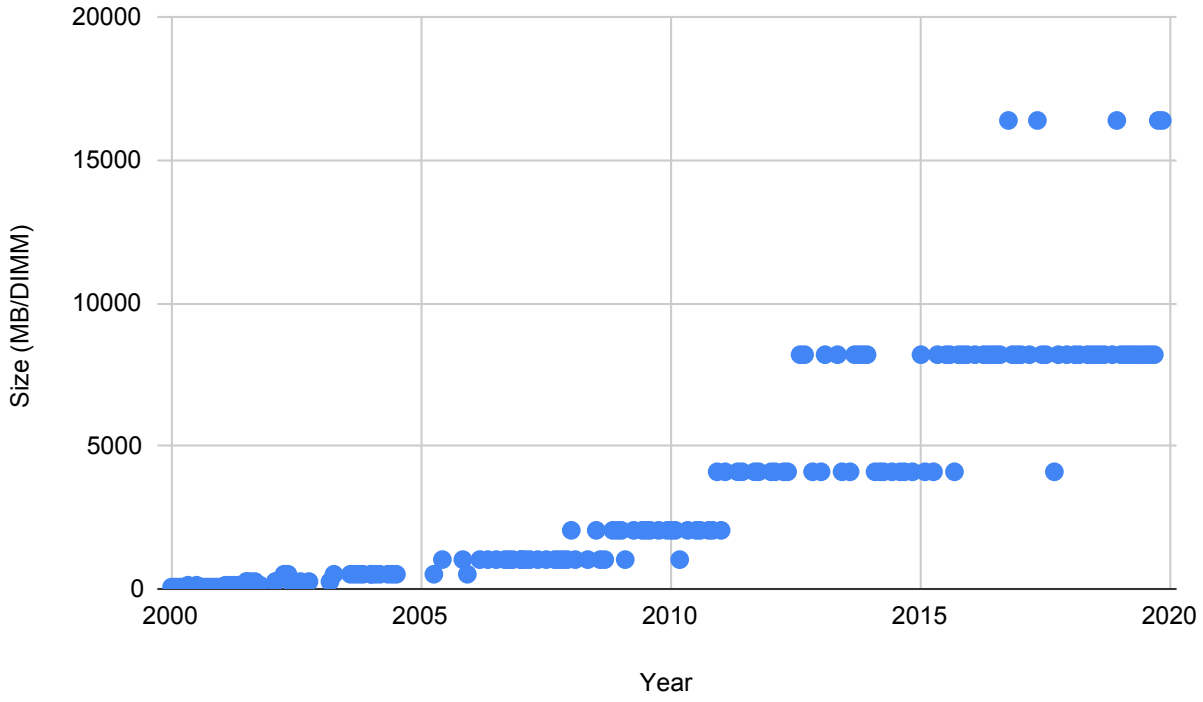


Figure 1.3: DRAM Size over Time

Google data center found that, when averaged over one-hour windows, at most 50% of available memory space was ever used. [33]. Utilization of memory bandwidth is similarly poor. Microarchitectural profiling of Google's data centers found that only 31% of the available DRAM bandwidth was used for 95% of the time (Figure 1.5). However, this same study found that memory utilization had a heavy tail, with utilization reaching a maximum of 68% [17]. It is this tail that makes good average utilization so difficult to achieve, as data centers must be provisioned for the worst case so that they can meet service guarantees.

1.3 Improving Utilization Through Disaggregation

One common method that data center designers use to improve utilization of resources is disaggregation. Ordinarily, each server has its own local pool of a resource. Jobs assigned to a node primarily use the resources available on that node while occasionally making RPC requests to other nodes. If a job exceeds the resource constraints of the node, it will either stall until other jobs finish and free up resources, or the job must be migrated to another node. As a result, server provisioning and scheduling must account for the worst case for each node. However, even when some servers are experiencing peak demand for a resource,

	MAINSTREAM MEMORIES				EMERGING MEMORIES		
			FLASH				
	SRAM	DRAM	NOR	NAND	STT-MRAM	PCRAM	RRAM
Cell area	>100 F^2	6 F^2	10 F^2	<4F^2 (3D)	6~50F^2	4~30F^2	4~12F^2
Multibit	1	1	2	3	1	2	2
Voltage	<1 V	<1 V	>10 V	>10 V	<1.5 V	<3 V	<3 V
Read time	~1 ns	~10 ns	~50 ns	~10 µs	<10 ns	<10 ns	<10 ns
Write time	~1 ns	~10 ns	10 µs–1 ms	100 µs–1 ms	<10 ns	~50 ns	<10 ns
Retention	N/A	~64 ms	>10 y	>10 y	>10 y	>10 y	>10 y
Endurance	>1E16	>1E16	>1E5	>1E4	>1E15	>1E9	>1E6~1E12
Write energy (J/bit)	~fJ	~10fJ	~100pJ	~10fJ	~0.1pJ	~10pJ	~0.1 pJ

Notes: F: feature size of the lithography. The energy estimation is on the cell-level (not on the array-level). PCRAM and RRAM can achieve less than 4F^2 through 3D integration. The numbers of this table are representative (not the best or the worst cases).

Figure 1.4: Comparison of SRAM, DRAM, FLASH, and NVRAM [41]

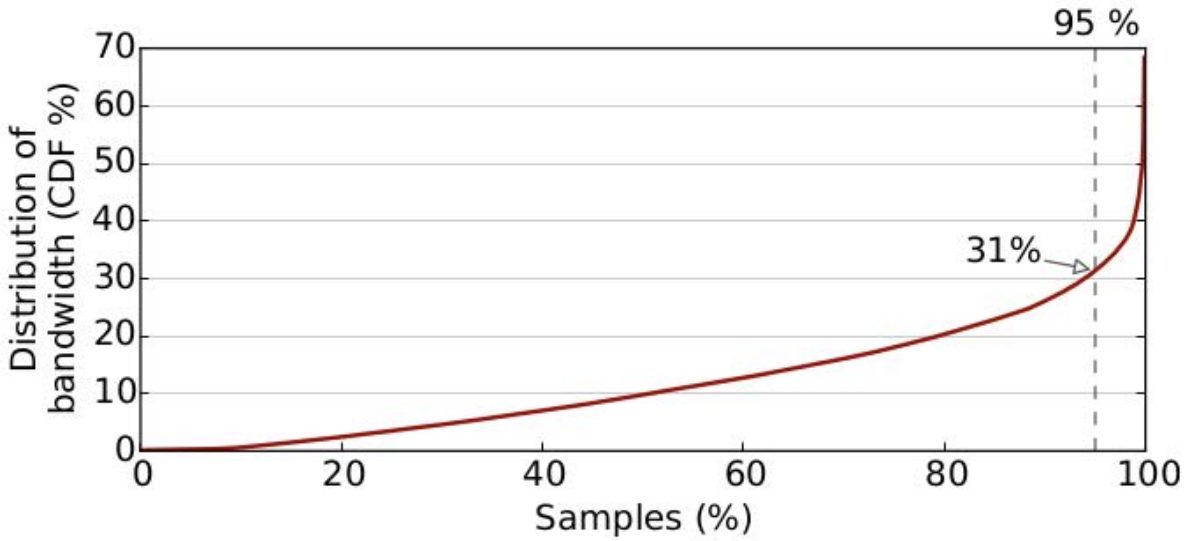
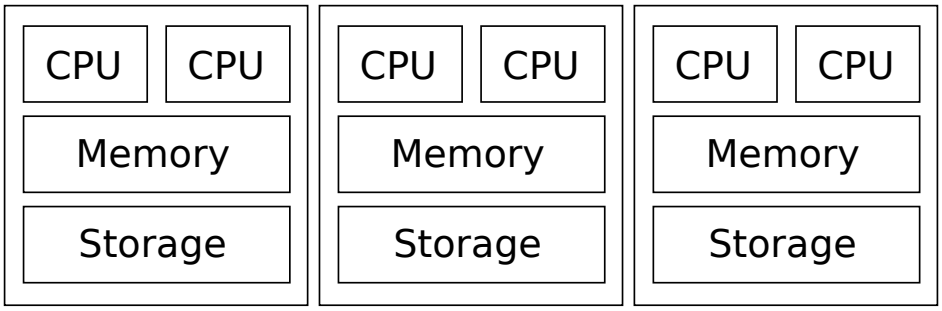


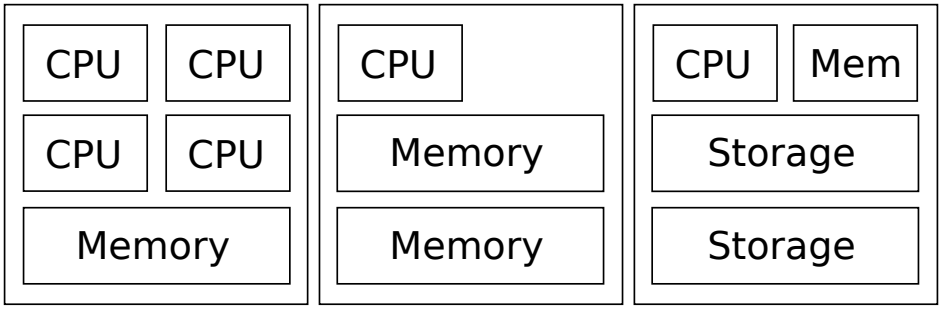
Figure 1.5: Memory Bandwidth Utilization [17]

other servers may have resources to spare. But with each server limited to using only its local pool, there is no way to reapportion the resources. This problem can be solved through resource disaggregation. In disaggregated systems, resources are moved off of the compute nodes and onto resource-specialized nodes. When jobs on the compute nodes need access to this resource, they make RPC requests to the specialized nodes. One compute node can request resources from multiple specialized nodes, and multiple compute nodes can share

resources on a single specialized node. Thus, a job with a higher resource requirement can easily request a greater share, while a job with lower requirements does not take up resources it does not need.



(a) Fully Aggregated Datacenter: Each node has identical configuration.



(b) Fully Disaggregated Datacenter: has CPU-specialized (left), memory-specialized (middle), and storage-specialized (right) nodes

Figure 1.6: Fully Aggregated vs. Fully Disaggregated Datacenter Organizations

1.4 Challenges of Implementing Disaggregated Memory

Such disaggregation has already been achieved for storage. Most data centers have moved their storage capacity from local disks and SSDs to dedicated network attached storage devices, file servers, or database servers. This is not the case with memory. Datacenter workloads with extensive memory requirements still mainly use locally attached DRAM. The reason memory disaggregation isn't as ubiquitous as storage disaggregation is because of the unacceptably high latency of conventional copper wire Ethernet networking. SSD latency is on the order of hundreds of microseconds, and disk latency is on the order of milliseconds, so the tens of microseconds latency of high-speed Ethernet is more tolerable. DRAM latency, on the other hand, is on the order of tens to hundreds of nanoseconds, so incurring the latency penalty of a network request for a memory access is far less manageable.

For that reason, the only common use for network-accessible memory currently is to cache results derived from database accesses, in which case the additional latency is still tolerable compared to reissuing the accesses to storage. However, a recent trend in communication technology has emerged which prompts a reevaluation of the feasibility of network-attached memory. Newly developed photonic interconnects transmit signals by sending light through optical waveguides instead of sending electric current through conductive wires. They have much lower latency and higher throughput than electrical interconnects, and with the development of integrated silicon photonics (fabricating photonics on the same IC as traditional semiconductors), using optical signaling to network data center nodes does not look very far off. Communication over photonic interconnect is on the order of hundreds of nanoseconds, so it is feasible to use it to connect compute and memory blades.

1.5 The Importance of Caching

Even with photonic interconnects, network latency will likely be at least an order of magnitude greater than that of locally-attached DRAM, which makes caching frequently accessed data in local DRAM crucial for maintaining application performance. There are many ways of performing such caching, which we will discuss in Chapter 2 and Chapter 3. This dissertation will focus on the design and evaluation of a hardware-managed DRAM caching system, in which all of the operations necessary to refill the local DRAM from remote memory are handled by a hardware controller rather than an application runtime or kernel fault handler.

1.6 Thesis Contributions and Overview

This thesis makes the following contributions:

- **Survey of existing remote memory implementations** - Chapter 2 presents a survey of previous remote memory systems, from early cache coherent non-uniform memory access (CC-NUMA) [40] [23] [1] [21] and cache-only memory architecture (COMA) [14] [10] [39] systems to modern virtual memory-based systems like Infiniswap [11] and Remote Regions [2]. Chapter 3 explains the weaknesses of virtual memory-based systems and why a hardware-managed DRAM cache would be a fruitful avenue for further research. It then describes the organization of existing on-package DRAM caches and how the techniques used in these systems may translate to DRAM caches for remote memory.

- **RTL implementation of remote memory system** - Chapter 4 explains why it is desirable to evaluate remote memory designs using RTL models and gives an overview of the various tools and libraries we developed in our lab to make RTL design easier, such as Chisel [4] and Rocket Chip [3]. Chapter 5 gives an overview of the FireSim platform [18], which we use to simulate network-connected RTL designs on FPGAs.

Using Chisel, Rocket Chip, and FireSim, we can construct detailed simulation models of multi-node disaggregated data center designs. Chapter 6 shows the design of a memory blade controller, which serves requests for remote memory entirely in hardware. It also shows the design of a remote memory client controller, which software can use to explicitly copy data between the local DRAM and remote memory. Finally, Chapter 7 shows the design of a hardware-managed DRAM cache, which can refill the local DRAM from remote memory in a way completely transparent to software and with very little overhead.

- **Realistic evaluation of remote memory system** - Chapters 6 and 7 demonstrate the raw performance of explicit RDMA-based and DRAM cache-based systems through simple applications and microbenchmarks running in FireSim simulations. Chapter 8 shows the results of more complex benchmarks using more realistic workloads. In particular, it shows a typical interactive workload in the form of the popular Memcached key-value store, as well as a typical batch workload in the form of a graph algorithm similar to those used by social networking applications.

Chapter 2

Background on Remote Memory Systems

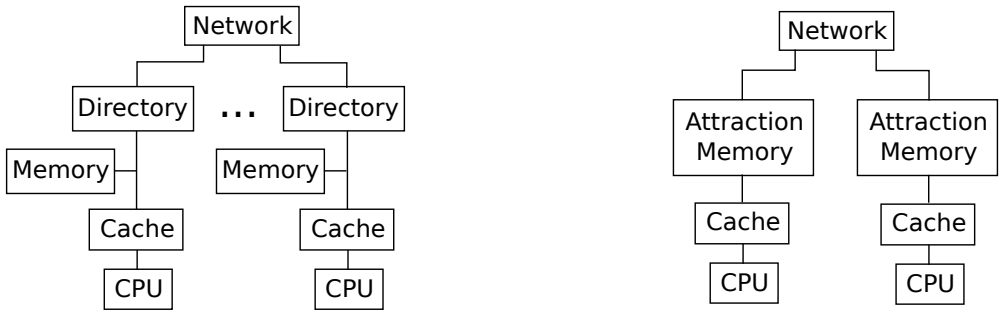
In the last chapter, we discussed how cluster-level disaggregation of volatile memory can improve memory utilization, thus allowing more cost-effective provisioning of data centers. Distributing memory across nodes or sockets in a cluster is a relatively old idea in computer architecture. In this chapter, we will discuss the various ways that disaggregated memory has been implemented in the past and how they compare to our approach. We will first look at early non-uniform memory access systems and then at more modern remote memory designs.

2.1 Early Disaggregated Memory Systems

The design considerations for a disaggregated memory system are similar to those of non-uniform memory access (NUMA) clusters that were designed in the 1990s. These systems were composed of pluggable modules, each holding one or two processors and a small amount of memory. When connected together, CPUs on one module could read and write memory on other modules, thus allowing the system to operate as if it were a single multicore computer. However, accessing memory on another module would be slower than accessing memory on the same module, making the access latencies "non-uniform". These systems can be further divided into two types: cache-coherent NUMA (CC-NUMA) machines, in which each node's main memory maps to a distinct portion of the address space, and cache-only memory architecture (COMA) machines, in which each node's memory is a cache and data can migrate from one node's memory to another's during execution.

Cache-Coherent NUMA Architectures

Some well known CC-NUMA designs are Hector [40], Stanford Dash [23], MIT Alewife [1], and SGI Origin [21]. The basic design of these systems are quite similar. All of them are



(a) CC-NUMA (b) COMA

Figure 2.1: CC-NUMA vs. COMA

composed of individual processor/memory modules connected together through an interconnection network or backplane. Each module contains a few processors and their associated caches, a small slice of main memory mapped at a fixed address, and a directory indicating which nodes are caching each line of the main memory. Cache misses for an address mapped to the local memory can be satisfied as it would in a single-chip system, but cache misses for addresses mapped to other memories will require a message be sent over the interconnect. If a CPU is performing a read, an entry for the requesting CPU will be added to the directory to indicate that it now shares the line. If a CPU is performing a write, the directory will be scanned for current sharers and invalidations will be sent to all the caches sharing the line.

One issue CC-NUMA systems must contend with is how to make the directory scalable as more nodes are added to the system. As more caches are added, more potential sharers must be tracked by each directory, which could require additional directory bits. One way to solve this issue is to recognize that each memory line will likely only be shared by a handful of caches. Thus, directory entries of a single bit for every cache can be replaced with a limited number of pointers. This is the approach taken by Alewife, which keeps four pointers in each directory entry. If the number of sharers exceeds four, the system falls back to software by sending interrupts to the home CPU for every coherence transaction. Even with this limitation, the directory still had considerable overhead, with a third of the memory on each node dedicated to the directory. Another method for reducing the size of the directory is to recognize that not all memory lines will necessarily be cached, so a limited directory cache can substitute for the full directory.

Another issue CC-NUMA systems must deal with is the performance impact of requests to non-local memory. For workloads that have good locality and no sharing between processes, this is not much of a problem, as the data is kept in a single node's cache and requests to non-local memory are infrequent. However, in cases where data is shared between caches on different nodes or where the problem size is larger than local memory, there could be a large number of requests to non-local memory, which degrades performance. To improve

performance in the former case, the system could provide ways to reduce invalidations by broadcasting updates to other caches upon writing to a shared cache line instead of invalidating their copies and requiring them to make new requests to memory. This is the method used by Dash, which provides `update-write` and `deliver` instructions that send updated data directly to the caches that are sharing the line. For the special case of shared locks or atomic variables, the system can provide in-place read-modify-write primitives that perform the atomic operations at the memory without requiring it to be read into the cache first and then invalidated. This is the approach taken by SGI Origin. In the case in which the working set is too large to fit in local memory, one way to hide the latency of long cache misses, assuming the application knows what memory will be accessed far ahead of time, is to use non-blocking prefetches. This primitive is provided by all of the CC-NUMA implementations cited. Another technique is page migration, in which a section of memory that gets frequent misses from a remote node is copied to that node's local memory. This can be done by the operating system using the virtual memory system, which is the approach taken by Alewife. Each page is given a miss counter, and when a threshold of misses is reached, an interrupt is sent to the CPU to trigger the migration.

Cache-Only Memory Architectures

Examples of COMA systems are the Data Diffusion Machine (DDM) [14] and the Kendall Square Research KSR-1 [10]. In these systems, the DRAM attached to each node is organized as an "attraction memory", which is essentially a cache. Like a cache, the attraction memory automatically brings requested lines into local memory. However, unlike a typical cache, there is no main memory backing the data. This means that, if a line is evicted from attraction memory and it is the last copy, it cannot simply be discarded as one would in a typical cache. Instead, it must be sent to another node to be placed in its attraction memory so that the data persists. This could potentially generate a lot of extra coherence traffic if the movement of the line triggers another eviction, so COMA systems must have some mechanism of designating space to limit the number of cascading evictions. DDM does this, for instance, by designating a "home" attraction memory for every line so that there is a canonical location at which an evicted line can be placed. As with CC-NUMA, the attraction memories in a COMA design contain both data and directory information. But the directories are not just used to find out which caches to send invalidations to on a write, but also to find out where to obtain a copy of the data on a read miss. The attraction memories are generally organized hierarchically, with each level of the hierarchy containing its own directory tracking which of the directories immediately under it possess each block. To find a specific line, a memory request starts at the attraction memory attached to the originating processor and ascends the hierarchy until it reaches a level that contains the requested line in its directory. It then uses the directory information to trace downwards through the hierarchy to an attraction memory that holds the actual data. This hierarchical organization allows some degree of spatial optimization by grouping processes that frequently share data in the same subtree.

A paper by Stenstrom, Joe, and Gupta [39] compares the performance of CC-NUMA and COMA implementations on different types of workloads. They break their workloads into five categories: workloads with few cache misses, workloads with mostly conflict/capacity misses, workloads with mostly compulsory misses, and workloads with mostly coherence misses. They found that workloads with few cache misses had similar performance between CC-NUMA and COMA, as most of the requests were satisfied by CPU caches. Workloads dominated by conflict/capacity misses were faster on COMA than CC-NUMA because misses in COMA were generally satisfied by the local attraction memory, whereas misses in CC-NUMA may be sent to a remote node. Workloads dominated by compulsory or coherence misses, however, were faster in CC-NUMA. For these types of misses, the desired cache line will not be in the local attraction memory, and since COMA systems have a deeper cache hierarchy and more complex routing, the latency of a miss is much longer than in CC-NUMA. The paper then highlights how each type of system can compensate for its weaknesses. For conflict/capacity misses, CC-NUMA systems can use page migration to move frequently accessed memory closer to the CPU. For compulsory misses, COMA systems can use prefetching to bring cache lines into the local attraction memory before they are accessed. And for coherence misses, COMA systems can use write-and-update type instructions to reduce ping-ponging of exclusive cache lines between attraction memories.

2.2 Modern Disaggregated Memory Systems

These CC-NUMA and COMA systems were designed at a time when the number of CPU cores and amount of DRAM that could fit on a single board were quite small compared to today. As compute and memory density improved, distributed CC-NUMA systems became obsolete. The processors and memory that would ordinarily be spread across a cluster could instead be aggregated onto a single board, with great improvements to latency and power efficiency. However, improvements to transistor and DRAM density have recently slowed (Fig. 1.3), which makes distributed memory systems again a topic of interest for designers seeking to expand the pool of memory available to data center workloads. There have been several proposed designs for such systems. These designs can largely be placed into four categories based on how they manage caching data in local DRAM: systems that require applications to explicitly copy data between local and remote memory, systems that manage local caching through the language runtime, systems that manage local caching through the virtual memory system, and systems that manage local caching in hardware.

Remote Memory with Explicit Data Movement

Some remote memory systems do not perform any sort of automatic local caching and instead require the application programmer to explicitly call library functions to copy data to and from remote memory. The simplest form of such a system would be key-value stores like Redis [34] and Memcached [9], which serve remote memory requests through standard

TCP/IP interfaces. The benefit of these systems is that they are easy to install in existing data centers, as they require no specialized hardware or operating system drivers. The downside, however, is high overhead. A memory request must traverse the software networking stack on both the client and the server. These traversals require many costly operations, like handling interrupts, context switches, and copying data between kernel and userspace buffers.

Systems that use remote direct memory access (RDMA), such as Infiniband and soNUMA [31], avoid these overheads through a zero-copy technique. As the name suggests, RDMA systems contain hardware controllers (henceforth called RDMA controllers) that can directly read and write from main memory and transfer data from main memory to RDMA controllers on other nodes. A "zero-copy" RDMA system is one in which the application interfaces with the RDMA controller directly, rather than going through a kernel driver. This means the controller can write the data directly to the userspace address instead of writing to a temporary kernel space buffer and then interrupting the CPU to copy the data to userspace. Additionally, one-sided RDMA operations allow transfers to be performed without involving any software on the server side. The server-side RDMA controller can read and write the requested memory address without having to interrupt the operating system to copy the data into a kernel buffer. The difference between TCP/IP and RDMA-based remote-memory access is illustrated in Figure 2.2. RDMA systems have the advantage of being relatively low overhead, but at the cost of a more complex API. Because the RDMA controller writes directly to a userspace buffer, the application code must explicitly manage various low-level aspects of the buffers, including concurrent access between the application and the RDMA controller.

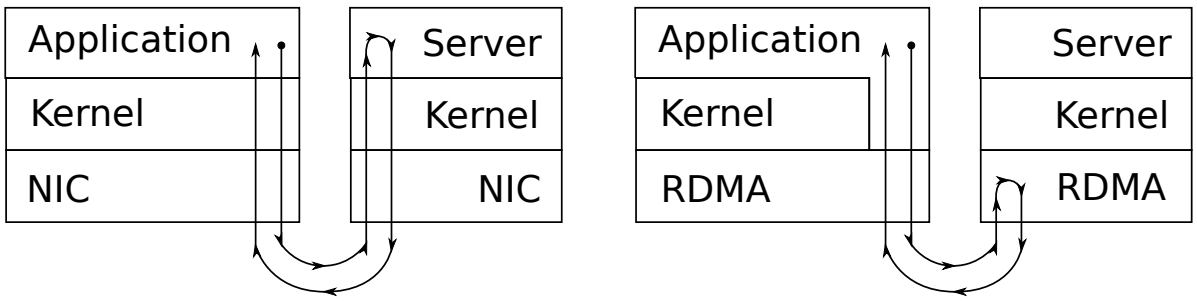


Figure 2.2: Remote memory access in TCP/IP (left) vs. RDMA (right)

Local Caching via Application Runtime

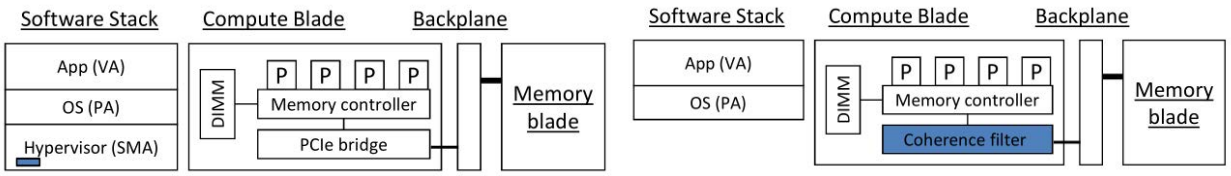
To manage the complexity of RDMA, many systems build abstractions on top of the RDMA API so that the application programmer does not have to reason about explicit transfers.

One way of abstracting RDMA is through the language runtime. This is the approach taken by languages like Unified Parallel C (UPC) [7]. UPC's programming model treats all of remote memory as a single partitioned global address space, with different parts of the address space assigned to different servers. Application code accesses this address space in the same way as local memory. The compiler inserts the necessary RDMA operations to keep the global address space consistent across nodes. The benefit of this system is that it is much easier to reason about than using RDMA directly. However, one remaining downside is that it requires all code to be written in a new language. This limits the ability to port existing code or integrate libraries written in other languages.

Local Caching via Virtual Memory

Another approach to building abstractions on RDMA is to use the operating system's virtual memory system to build a page cache. Virtual memory is a feature provided by the memory management unit (MMU) of the CPU that translates virtual addresses used by applications to physical addresses in main memory. This is done by dividing virtual memory into fixed-size pieces (usually 4096 bytes each) called pages and mapping each page to a corresponding region of physical memory called a frame. Virtual memory is generally used by the OS kernel to isolate user processes from each other so that they can use the same virtual addresses but different physical addresses. The kernel can also use the virtual memory system to make it appear to the application that main memory is larger than it actually is. If there is not enough physical memory for all the virtual memory requested, some pages can be written to secondary storage instead. When a page not present in main memory is accessed by the application, the MMU triggers an exception called a page fault, which invokes a page fault handler to read the data back from secondary storage. This technique is called a page cache, as it makes the main memory a cache for secondary storage. Page caches can be used for remote memory simply by storing the additional pages in remote memory instead of secondary storage and having the page fault handler send RDMA requests rather than disk/SSD reads.

A paper by Lim et al. [25] explores such a design. This paper proposes a design for a memory blade that attaches to the same backplane as the compute blades through PCI Express or HyperTransport. It has a hardware controller that handles read and write requests directly in hardware. Their memory blade design abstracts the physical memory by translating what it terms system memory addresses (SMA) from the compute blades into remote machine memory addresses (RMMA), which are used to address the DRAM DIMMs. Using a virtual memory system like this allows the memory blade to perform allocation using a similar scheme to regular operating systems. Management software running on the memory blades maps superpages from SMA to RMMA. The authors evaluate two different variations using trace-based full system simulation. The first is a page-swapping (PS) system (Fig. 2.3a), in which the memory blade is treated like a swap device. The OS on the compute blade sets up a mapping from virtual memory to remote system memory. On an eviction, the page is written to the memory blade and then read back when the page is swapped back



(a) Page Swap (PS) (b) Fine-Grained Remote Access (FGRA)

Figure 2.3: PS vs. FGRA remote memory system designs [25]

in. They also evaluate a second system they evaluate is a fine-grained remote access (FGRA) system (Fig. 2.3b) that allows remote memory to be accessed at cache line granularity. In this system, requests for remote memory get sent to a coherence filter. The coherence filter absorbs all cache coherence requests except for reads and writes, which get forwarded to the memory blade. Though the FGRA system allows finer grained access, it has the major downside of virtually all operations resulting in remote memory accesses.

The Infiniswap paper [11] implements disaggregated memory as an abstraction on top of Infiniband RDMA. It does not separate servers into compute blades and memory blades. Instead, it provisions both an Infiniswap Block Device and an Infiniswap daemon on each server. The block device uses a kernel module to expose a conventional block device interface to the application. This block device can be explicitly read from and written to or used as a swap device. The "storage" for this block device is composed of fixed-size slabs of RDMA-accessible memory distributed across different hosts. The block device uses one-sided RDMA reads and writes to access the remote memory for the slabs it manages, so the processor on the remote node is not involved in the transfer. The Infiniswap daemon runs entirely in userspace and handles just the control plane operations (i.e. allocations) for remote memory. It takes allocation requests for slabs from the block devices. It can also preallocate and proactively evict slabs to improve latency. Allocations are performed in a decentralized manner by using a "best-of-two" randomized approach. The daemon contacts two daemons on remote nodes and picks the one with the lower memory usage to request an allocation from. This distributes allocations evenly across the cluster without creating a bottleneck by having a central allocator.

The Decibel paper [30] implements a disaggregated storage system using non-volatile memory. Decibel splits the server node into per-core shared-nothing runtimes, each of which has exclusive access to a NIC and NVMe queue. Each runtime manages a collection of decibel volumes (dVols), which virtualize the NVM. Each dVol presents a sparse virtual address space of fixed-size blocks, which get mapped to non-contiguous physical addresses on the NVM. Since the per-core runtimes share the same NVM, they need to coordinate with each other to split up the physical address space. Decibel amortizes this synchronization cost by splitting reservation from allocation. Each per-core runtime maintains a pool of reserved blocks, which it allocates to the dVols under it. When the pool runs low on unallocated blocks, the runtime

expands it by reserving more from the NVM in large, multi-block extents. The key feature of Decibel is its performance isolation guarantees. Users can specify service-level objectives (SLO) for each dVol, which guarantees a certain bandwidth. Per-core runtimes have a fixed request window of outstanding requests sent to the NVM. They share this request window among dVols by scheduling requests according to the SLO set for each.

The remote regions paper [2] is another abstraction on top of Infiniband. The interface is quite similar to Infiniswap. It provides a virtual filesystem called RegionFS. Applications can create files in RegionFS and map them into memory. All of the servers in the RegionFS cluster share the same namespace, so if two applications on different hosts open files with the same name, they will be able to share memory. RegionFS pages that are mapped into memory are cached locally in a page cache. Page faults are used to bring data into the cache the first time it is read and to mark the page dirty on the first write. The RegionFS library provides prefetch and mark-dirty functions to trigger these faults ahead of time so that the latency isn't incurred during execution. There is no coherence, but the API provides functions to explicitly flush or clear the cache. The system is designed in four parts: a user library, a RegionFS file system, a daemon, and a manager. The user library is linked to the application and provides the API for interacting with regions. The file system is a VFS module that implements all the filesystem operations that the user library uses. It maintains the list of open regions, which keeps track of where to find the data for each region. The daemon is what directly manages the remote memory. It exports the memory in big pools to be accessed over RDMA and makes regions persistent. The manager handles all control operations like creation, opening, closing, deletion, and memory-mapping of regions and allocation of memory.

A paper by Koh, Kim, Jeon, and Huh [20] takes a slightly different approach from the others. Instead of abstracting remote memory as a block device and using OS page faults to pull data into local memory, their design instead pushes remote memory handling down to the hypervisor. The RDMA operations are conducted by a specialized handler instead of the typical page fault routines, thus avoiding some of the overhead of these routines. The hypervisor handler can hide some of the latency of remote memory interactions by overlapping RDMA requests with computation. For instance, by allowing the VM to execute on a fetched page as the victim page it is replacing gets written back. Their design also uses a novel technique called elastic block management, in which consecutive pages can be merged together into a larger block to take advantage of spatial locality. If, on eviction, less then half of the pages in a merged block have been used, it gets broken up again into individual pages. Otherwise, the entire block will be fetched together the next time one of its constituent pages is accessed.

The advantage of these virtual memory-based systems is that the application programmer can write code treating remote memory as if it was local memory. And since the RDMA operations are handled by the operating system, rather than a language runtime, the applications can be written in any language. The main downside of such systems, however, is that relying on the kernel's virtual memory system can introduce large overheads, which we shall demonstrate later in Chapter 6.

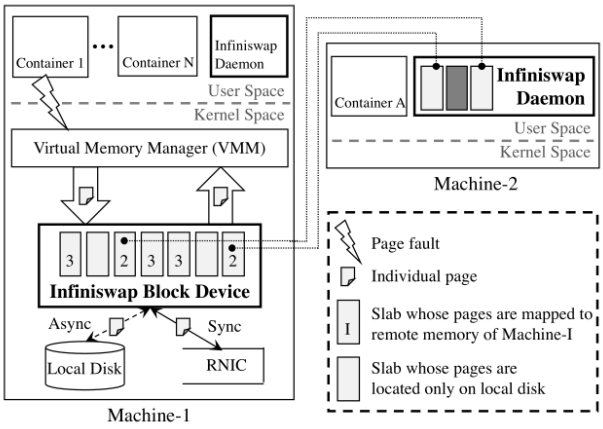


Figure 2.4: InfiniSwap Architecture [11]

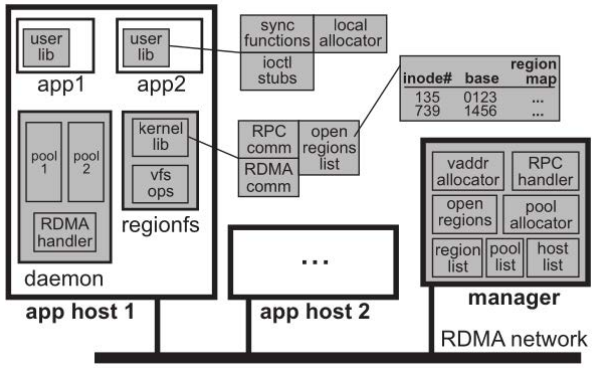


Figure 2.5: Remote Regions Architecture [2]

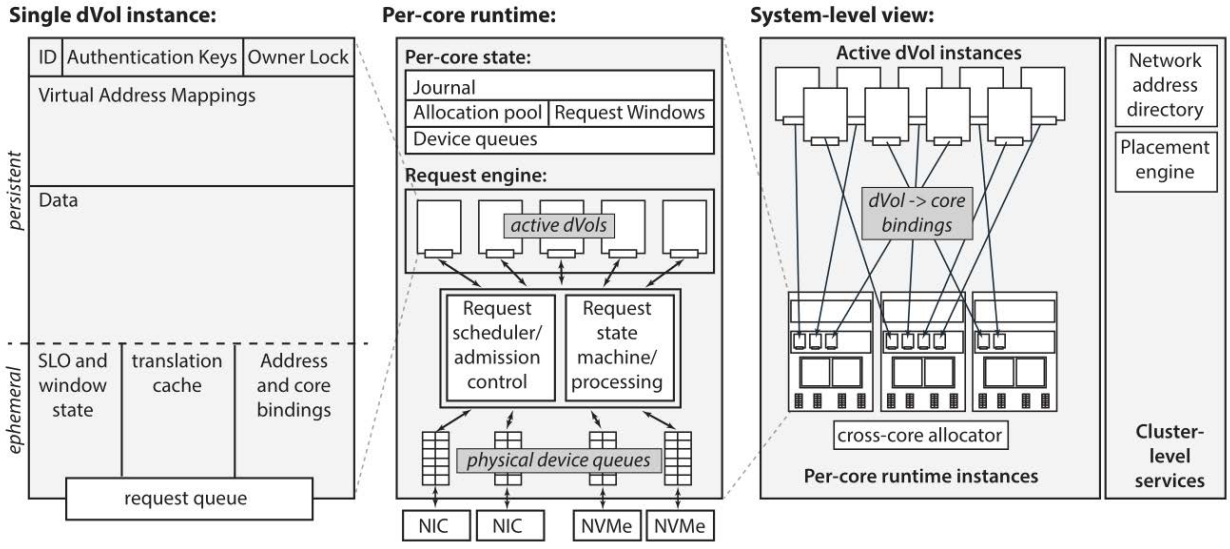


Figure 2.6: Decibel Architecture [30]

Hardware-managed Local Caching

The final category of remote memory systems are those that manage local caching entirely in hardware. This means that the RDMA requests needed to refill the cache on a cache miss are issued directly from hardware. This maintains the advantages of virtual memory-based systems in making cache misses transparent to the application programmer while reducing the high cost of invoking a page fault.

An example of such a system is outlined in the LegoOS paper [36]. LegoOS is a fully disaggregated operating system in which the different subsystems — compute, memory, and storage — all run on different nodes and are managed by logically independent monitors. They term this approach to operating system design a "splitkernel" approach. The compute blades, which they term pComponents, contain the CPUs, L1, L2, and L3 caches, and a software-managed DRAM cache they term an ExCache. This is essentially equivalent to the page caches used in Infiniswap and RegionFS. All of the caches are virtually indexed and virtually tagged. The pComponent has no knowledge of physical memory. On the pComponent runs the compute monitor, which is responsible for scheduling threads and managing the ExCache. If there is a miss in the ExCache, the compute monitor sends a request to the memory blade, which they term the mComponent. The mComponent contains all the hardware involved in satisfying a memory request: TLB, MMU, and DRAM. The mComponent runs a memory monitor, which is responsible for allocating memory and setting up page mappings. Reads and writes to memory bypass the software and access memory directly through RDMA. The decision of which pComponent to schedule a process on or which mComponent to allocate memory on is handled globally. Since this creates bottlenecks, LegoOS amortizes the costs by performing the allocation more coarsely and leaving finer scheduling/allocation decisions to individual components. For instance, processes are assigned to pComponents globally, but all threads that spawn from a process are scheduled on the same pComponent. Virtual memory is assigned to mComponents in 1GB sized vRegions, which then allocates it page by page. For their evaluation, the authors emulated the different components on commodity servers by disabling the parts of the server that weren't being used for a component. The pComponent had most of its memory disabled except a small amount used for the ExCache. The mComponent had all of its processors disabled except one, which was used to run the memory monitor. Communication between pComponents and mComponents were carried over Infiniband.

Hardware-managed systems combine the ease-of-use of virtual memory-based systems with the low overhead of explicit RDMA-based systems. Their main disadvantage is that they require custom hardware, which means data center operators would need to purchase new equipment to take advantage of them. However, since programs that use only local memory can be easily made to use remote memory, data center operators could potentially slowly move their systems to using remote memory as part of the regular hardware replacement schedule.

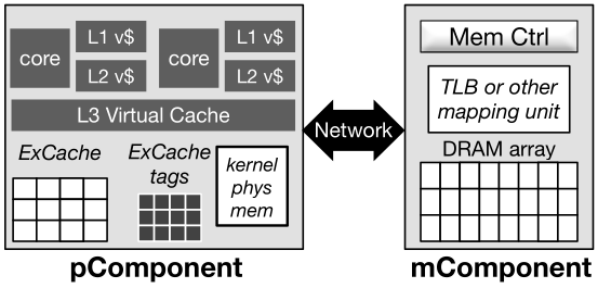


Figure 2.7: Lego OS pComponent and mComponent Architectures [36]

2.3 Chapter Summary and Discussion

In this chapter, we have discussed the organization of various existing disaggregated memory systems and categorized them based on how they handle the issue of local caching. This includes explicit transfers in application code, caching mediated by the language runtime, caching mediated by the operating system, and caching managed entirely by hardware. These different categories represent tradeoffs among three major considerations: performance, ease-of-use, and cost. Systems that use explicit RDMA transfers have low overhead but are difficult to use. Systems with virtual-memory managed caching are relatively easy to use, but have high overhead. Systems with language runtime managed caching are somewhere between explicit RDMA and virtual memory systems in terms of ease-of-use and performance. Hardware-managed local caching has both low overhead and is easy to use. However, it is the only one out of the four to require custom hardware, which makes it far more costly to implement. Perhaps owing to cost considerations, such systems do not yet see significant commercial use and are not as well-studied in academia. Of the various recent papers we surveyed, only the LegoOS paper proposed a hardware-managed DRAM cache, and even that paper resorted to emulating the hardware using software page fault handlers. This is likely because hardware design has traditionally been much more labor-intensive and time-consuming than software development. In Chapter 4, we will discuss the tools we have built to make hardware design more productive, thus making it feasible for a small research team to undertake a complex task like designing a DRAM cache controller. But first, in the next chapter, we will look at existing DRAM cache systems, which use on-package DRAM as a cache for off-package DRAM, and see how the techniques used can be applied to DRAM caches for remote memory.

Chapter 3

Background on DRAM Caching

As we saw in Chapter 2, most systems designed to use remote memory cache data in local DRAM in some way. Given relatively long network latencies (even with next-generation photonic interconnects), such local caching is necessary to maintain good performance. Usually, this involves using virtual memory to implement an OS-managed page cache, as is done in the remote regions paper [2], the InfiniSwap paper [11], and the PS system in the Lim, et al. paper [25]. The advantage of this approach is that no hardware alterations must be made on the compute blade, but application code can still run mostly unmodified. The downside is that a cache miss will be more expensive, as it will now incur latency from the OS page fault handler in addition to the network request latency. Furthermore, context switching from the page fault handler ends up degrading the performance of compute-intensive applications, as the InfiniSwap authors discovered with their VoltDB benchmark. Using the virtual memory system also means the size of a "cache line" is constrained by the system page size (usually 4K). This can lead to wasted bandwidth, as a page is brought in from remote memory, but only a fraction of it is used. The Lim et al. paper suggests a different approach through their fine-grained remote access system. Using FGRA alone would cause things to slow down, since every memory access would go to remote memory. To mitigate this, they implement a page migration scheme so that frequently accessed pages are migrated to local DRAM instead.

A variation on local caching for remote memory that doesn't seem to have been thoroughly explored is a hardware-managed DRAM cache. DRAM caches are well-studied and multiple designs have been researched and manufactured. However, existing DRAM caches mainly use high-bandwidth stacked DRAM as a substitute for SRAM to produce last-level caches backed by off-package DRAM. This affects key design decisions like the cache line size, the location of metadata, and the tradeoff between access latency and hit rate, which we will discuss in this chapter.

3.1 Cache Line Size and Location of Metadata

One major design tradeoff that needs to be considered for DRAM caches is between large cache lines (typically around 4 kB) and small cache lines (typically 64 B). As with SRAM caches, a larger cache line can take more advantage of spatial locality and reduce the number of compulsory misses, but at the expense of bandwidth wastage if not all data in a cache line is used before eviction. For DRAM caches (and large caches in general) there is the additional consideration that smaller cache lines incur a larger storage overhead for metadata. As DRAM caches can be several gigabytes in size, storing separate tags for each 64-byte block could lead to a very large metadata overhead. This leads to another tradeoff that must be considered, which is whether to store metadata in SRAM or in the DRAM alongside the data. Storing metadata in SRAM leads to quicker cache lookups, but is very expensive in terms of area, thus requiring larger cache lines. Keeping metadata in DRAM can allow more to be stored, thus enabling smaller cache lines, but at the expense of a longer hit latency.

Sub-blocking

One way of balancing tag storage overhead and bandwidth usage is to use sub-blocking, in which a larger cache line is broken up into smaller sub-blocks. The cache stores a single tag for the entire block, but individual valid and dirty bits for each sub-block, which makes the metadata storage overhead similar to that of a cache with the larger block size. On the initial miss for a cache line, a tag is allocated in the metadata array, the specific sub-block that was accessed is fetched into the cache, and its valid bit is set. Subsequent accesses to the block will pull in the other sub-blocks. When a cache line is evicted, all of the sub-blocks present are removed and the dirty sub-blocks are written back.

A variation of this technique is used by the Footprint Cache [16], which uses 4 KB blocks and 64 byte sub-blocks. The use of page-sized cache lines allows the cache to store metadata in SRAM. One difference between the footprint cache and a standard sub-blocked cache is that the footprint cache fetches more than one sub-block on the initial miss. This set of blocks is called a "footprint" and is predicted by a footprint predictor table, which is indexed by a combination of the address of the instruction that triggered the access and the offset of the accessed address within the cache line. The rationale for this design is that a given instruction tends to access memory in a similar pattern every time it is invoked. On a miss to a new block, the predictor table is read and only the requested sub-block and the sub-blocks in the predicted footprint are fetched. The predictor index is stored in the metadata array so that, when the cache line is evicted, it can be used to update the predictor with the actual footprint of the cache line (i.e., the sub-blocks that are actually valid in the cache line before eviction). One other optimization the footprint cache makes is to detect singleton cache lines, lines with a footprint consisting of only a single sub-block, and not allocate them in the DRAM cache. This reduces the amount of wasted space in the cache, as the entire cache line would need to be allocated otherwise, even though only a single sub-block is valid.

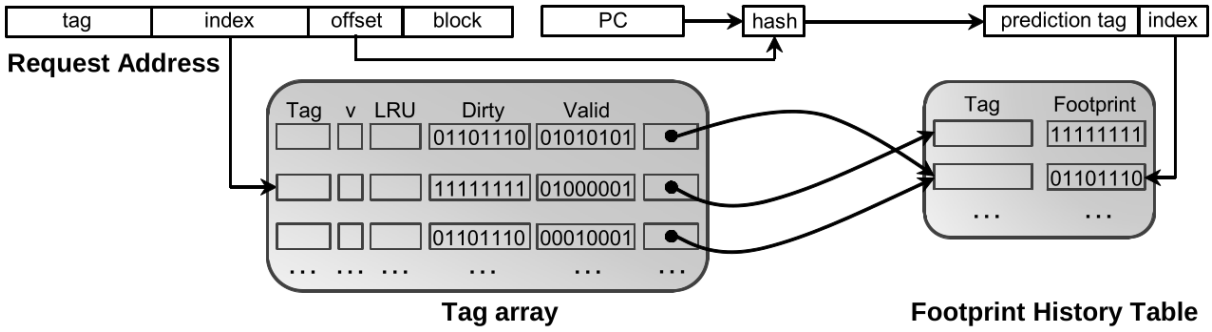


Figure 3.1: Footprint Cache tag array and footprint history table [16]

The downside of sub-blocked caches like Footprint Cache is the potential for wasted space, as each cache line takes up the full space regardless of how many sub-blocks are actually present. The Footprint Cache mitigates this to some extent by excluding cache lines predicted to only have a single valid sub-block from being cached. However, there is still potential wasted space from lines with a small (but greater than one) number of valid sub-blocks. A paper proposing a bi-modal DRAM cache [12] addresses this issue by instead allowing the cache to use variable-sized blocks. Instead of all cache lines being large lines that gets subdivided into sub-blocks, the bi-modal cache organizes each cache set to have a mix of large and small cache lines. The small cache lines can all be allocated and evicted independently of each other and do not need to be part of the same larger cache line. The bi-modal cache predicts whether a miss should be filled with a large or small cache line based on previously expressed spatial locality. The cache keeps utilization vectors for a sample of sets, which tracks which 64-byte sub-blocks in the set have been utilized. It also maintains a table of saturating two-bit counters indexed by a subset of the cache tag and index. On eviction, the counter is incremented if the number of set bits in the utilization vector exceeds some threshold and decrements the counter otherwise. This way, the cache will only allocate larger cache lines when the line is likely to be filled. One disadvantage of this system compared to sub-blocking is that it needs to maintain individual tags for the smaller blocks, which means it can no longer keep tags in SRAM. It uses a technique for mitigating the latency of tags-in-DRAM that we will discuss in the next subsection.

Reducing Tags-in-DRAM Latency

A different path to improving performance is to use small cache lines and mitigate the latency overhead of tags-in-DRAM. DRAM metadata lookup overhead can be separated into two cases: overhead for DRAM cache hits and overhead for refills on cache misses.

The latency on cache hits can be mitigated by interleaving the fetching of metadata with fetching the cached data itself. One technique used in a paper by Loh and Hill [26] is to take

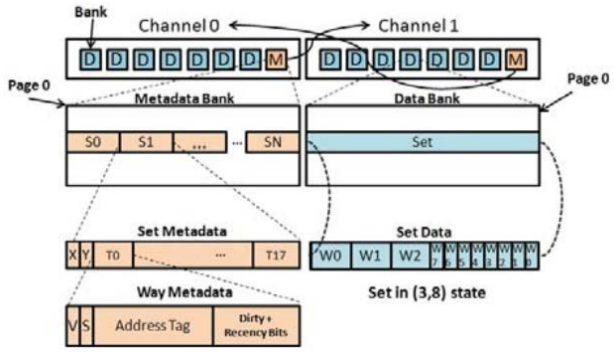
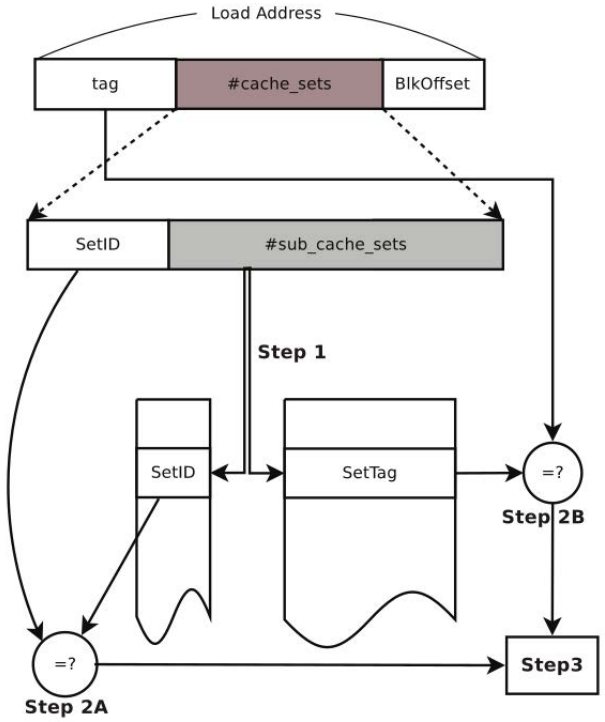


Figure 3.2: Bi-modal Cache data and tag layout [12]

advantage of DRAM row locality and pack metadata and data together in a single DRAM row. When the metadata is read, the full row is read into the DRAM row buffer. Then, after the metadata is checked to determine which way the cache line is in, the data itself can be read from the row buffer without paying the penalty of an additional row activation. If the metadata needs to be updated (e.g. by setting a dirty bit), the write can also be performed to the row buffer. While this method avoids an extra row activation, it still pays the penalty of an extra column access. The Alloy Cache [32] paper avoids this by streaming metadata and data from the DRAM together in a single compound access. However, for this to be possible, the cache must be direct-mapped, which could potentially lead to more conflict misses. The paper's authors argue that the increase in miss rate is not large enough to override the benefits from reduced hit latency. A side-effect of this organization is that the number of sets in the cache is not a power of two (since each DRAM row fits 28 cache lines), which means modular arithmetic is needed to determine the index for a given address.

Alternatively, instead of interleaving the metadata and data access, the cache could skip accessing DRAM for metadata in certain cases by caching metadata for some lines in SRAM. This is the strategy taken by ATCache [37], which maintains a direct-mapped SRAM cache of tags. The cache is indexed by a portion of the set index. The remaining bits of the index are matched against a SetID stored in the SRAM cache. If the SetID and the stored tag match the requested address, the request can be determined to be a hit and access to DRAM for the metadata can be skipped. Otherwise, the DRAM is accessed to scan the full metadata for the set. As a further optimization, ATCache can take advantage of spatial locality by prefetching metadata for some number of subsequent cache lines into the SRAM cache on a miss. The bi-modal cache described in the previous subsection uses a similar method, maintaining a two-way set-associative cache for metadata which they term the "way locator". They chose to cache the two most recently used ways because they found that, on average, 94% of hits to the cache are for these ways. The way locator is also used as part of the cache replacement policy. If the set the cache line is being evicted from is present in the way locator, one of the ways other than the two kept in the way locator is randomly

evicted. If the set is not present in the way locator, any of the ways could be randomly evicted. In this way, the design generally avoids evicting the most-recently used cache lines without needing to update the metadata for every access.



Step 1	Locate SetID and SetTag in AT-Cache by a subset of #cache_set (#sub_cache_sets).
Step 2A	Check SetID to determine if AT-Cache set contains SetTag.
Step 2B	Check SetTag to determine if DRAM cache contains requested data (DRAM cache hit/miss).
Step 3	If step 2A is hit, use step 2B's result to determine hit/miss in DRAM cache. Otherwise, issue compound access to DRAM cache.

Figure 3.3: ATCache access logic [37]

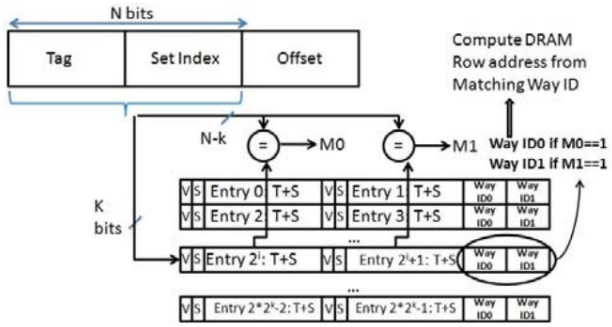


Figure 3.4: Way Locator [37]

To reduce latencies on miss, the cache can predict whether or not the request is a miss and issue the request to main memory simultaneously with the request to the cache DRAM

for the full metadata. This is the strategy taken by the Loh and Hill paper, which uses a structure called the "Miss Map" to track which blocks are present in the cache. Each entry in the MissMap tracks the blocks within a large page-size region and is composed of a single tag and a bit-vector with one bit for each cache line. This is similar in structure to the SRAM metadata array for a cache with large sub-blocked cache lines, but unlike a sub-blocked cache, these entries only determine whether a block is present and not its location within the cache or whether or not it is dirty. While this eliminates the latency overhead of a metadata in DRAM lookup on miss, it imposes a limitation on what cache lines can be kept in cache. The MissMap cannot mispredict, as this could cause stale data to be read from main memory into the cache. As a result, if an entry is evicted from the MissMap, the cache lines tracked by that entry must also be evicted to ensure the MissMap accurately reflects the state of the cache. If any of those cache lines are dirty, they must be written back to main memory. The MissMap structure is also fairly large and consumes a large amount of SRAM. The authors propose taking resources from the L3 cache to provide space for the MissMap.

A paper by Sim et al. [38] relaxes the restriction that the miss predictor be 100% accurate. Like the MissMap, their predictor keeps an entry for a larger multi-line region. But instead of keeping a bit vector with precise information, they instead store a two-bit saturating counter that is incremented on a hit and decremented on a miss. This greatly reduces the amount of space needed, but also means that both false positives and false negatives are possible. Dealing with false positives is straightforward. If the predictor erroneously predicts that a line not yet present in the cache is a hit, the error will be discovered once the full metadata is accessed. The refill request can then be sent out, resulting in only a performance penalty and no correctness issue. False negatives are trickier to handle. Issuing a refill request when the cache line is actually already present and potentially dirty risks overwriting the cache line with stale data. Sim et al.'s design deals with this issue by operating most regions of the cache with a write-through policy and only using write-back for regions with high write traffic. That way, regions that are marked write-through can speculatively issue refill requests and return data from main memory to upper-level caches without the risk of overwriting dirty data.

3.2 Optimizing Latency vs. Optimizing Hit Rate

Since the stacked DRAM isn't that much faster than off-package DRAM, there is a much greater focus on reducing overall latency and power consumption than on maximizing the hit rate. Therefore, many of the systems described in the previous section make design decisions that sacrifice better hit rates for lower latencies. For example, as mentioned previously, the Alloy Cache uses a direct-mapped cache rather than a set-associative one so that it can simultaneously read cache data and metadata. Other caches use random replacement instead of some variant of LRU replacement to avoid updating the metadata in DRAM on every access. The system designed by Sim et al. goes even further and sometimes purposefully

requests data from main memory that is predicted to already be in the cache. This is done to balance the bandwidth usage between the stacked DRAM and off-package DRAM and is hence called "self-balancing dispatch". Despite increasing the number of misses, this technique does improve the performance as a whole. The reason it works is in the formula the cache uses to decide whether a predicted-hit request should be dispatched to the DRAM cache or main-memory. The system tracks $N_{off-package}$ and $N_{dram-cache}$, the number of requests waiting for off-package memory and stacked DRAM, respectively. It then multiplies each number by the latency of a single request to determine the expected queuing delays $E_{off-package}$ and $E_{dram-cache}$. If $E_{off-package} < E_{dram-cache}$, the request is sent to the off-package memory rather than the cache DRAM, as its expected completion time would be shorter, even with a longer delay on the wire.

3.3 Chapter Summary

In this chapter, we have surveyed various techniques used and tradeoffs considered in the design of stacked DRAM caches for off-package DRAM. The primary tradeoff when designing DRAM caches is between latency and efficiency. Systems using large cache lines reduce the average latency by using SRAM metadata storage and amortizing miss penalties through spatial locality. However, large cache lines can lead to wasted space and bandwidth if there is significant fragmentation. Meanwhile, small cache lines may have longer latencies from storing metadata in DRAM and more frequent compulsory misses, but the more granular access ensures that bandwidth and storage are more efficiently used. To evaluate this tradeoff, we will design our hardware generator to make the cache line size configurable. In the next chapter, we will discuss the tools we have developed that make building such generators possible. Then, in Chapter 7, we will show the results of our design space exploration.

In our literature review, we also found many different methods of reducing the overhead incurred from metadata lookups, either by moving all or some metadata into SRAM or interleaving the metadata lookup with other work. For caching remote memory, using an SRAM cache as in [37] may be useful. Making use of DRAM row buffer locality, as in [26], would not be so useful, however, as it requires keeping the DRAM row open between reading the metadata and refilling the cache data. This means that there could only be one request in-flight per DRAM channel, which is simply not acceptable if the DRAM cache is to achieve good refill bandwidth. Also out of the question is hit-miss prediction or self-balancing dispatch. Accessing off-package DRAM can take 10s or 100s of nanoseconds, but even with fast photonic networks, requesting data from remote memory could potentially take a microsecond or more. Therefore, sending a remote memory request unnecessarily (either speculatively or deliberately) could only degrade performance. In Chapter 7, we will show the results of a design-space exploration on whether SRAM caching of metadata is effective at improving performance.

As mentioned in the last chapter, one reason hardware-managed DRAM caching for remote memory hasn't been explored as much as virtual memory-based systems is because

Parameter	Technique	Benefits	Drawbacks
Cache Line Size	Small Lines	Less wasted bandwidth Fewer conflicts	Large metadata overhead Poor spatial locality
	Large Lines	Small metadata overhead Good spatial locality	More wasted bandwidth More conflicts
	Sub-Blocking	Less wasted bandwidth Smaller metadata overhead	Worse spatial locality More conflicts More wasted space
	Bi-modal Cache	Less wasted bandwidth Good spatial locality	Large metadata overhead
Metadata location	Tags in DRAM	Lower area/power/cost	Longer latency
	Tags in SRAM	Lower latency	Higher area/power/cost
	SRAM tag cache	Lower avg. latency	Larger area
Metadata / Data Layout	Metadata and data in same DRAM row	Lower latency	Limits bandwidth
	Metadata and data in single access	Lower latency	Limits associativity Irregular line size
Access Parallelism	Miss Map	Lower avg. latency	Larger area More evictions
	Miss Prediction	Potentially lower latency Lower area overhead	False misses Requires write-through
	Self-Balancing Dispatch	Balanced bandwidth usage	Requires write-through

Table 3.1: Pros and Cons of Various DRAM Caching Techniques

designing custom hardware and simulating a multi-node cluster has generally been too time-consuming and expensive for a small team of researchers to accomplish. In the next chapter, we will discuss the tools we have developed to make designing custom hardware easier and more agile. Then, in Chapter 5, we will discuss FireSim, a cloud FPGA-based simulation platform that allows for fast, cycle-accurate simulation of multiple network-connected custom hardware nodes.

Chapter 4

Hardware Generators

In Chapter 2, we saw various designs of remote memory systems. All of these designs were either software-based systems that ran on commodity hardware or, in the case of [36], novel hardware designs emulated using existing hardware. To conduct a higher quality performance analysis of remote memory systems using novel hardware designs, we need some way of simulating the hardware in a timing-accurate way. In Chapter 3, we saw some studies which did design new hardware structures for DRAM caches. Most of these studies simulated the new hardware using abstract functional simulators like Gem5. While such simulators are useful for quickly exploring new architectural ideas, they cannot fully capture detailed microarchitectural constraints that might affect the feasibility of the design. They also cannot be used to determine power or area cost. To get this sort of information, the design has to be implemented in RTL, but RTL development can be very time-consuming, especially if multiple design points must be implemented and evaluated for a design space exploration. In our lab, we have designed a hardware description language, Chisel, and library of RTL components that make RTL design more productive and allow multiple designs to be elaborated from a single codebase.

4.1 The Chisel Hardware Design Language

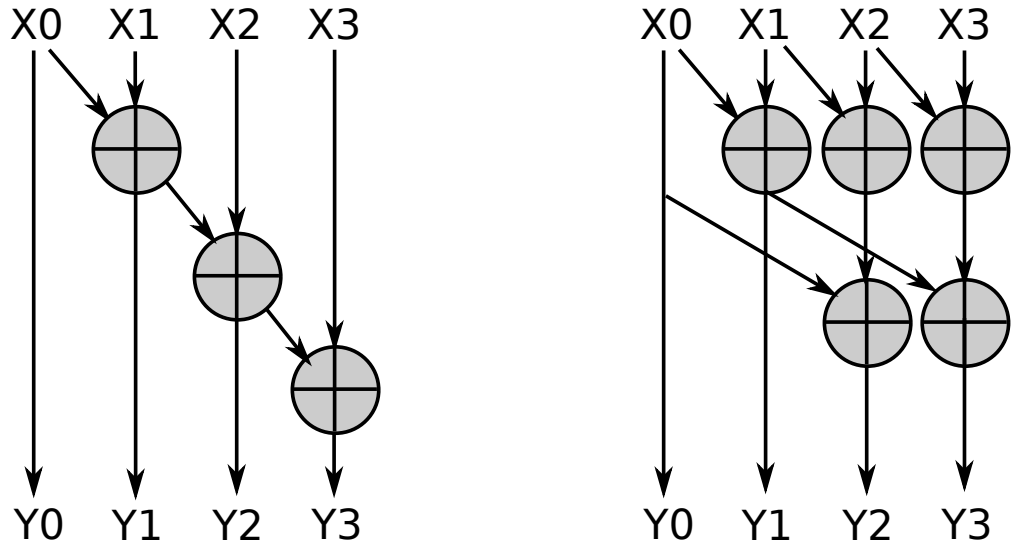
Chisel [4] is a hardware description language (HDL) written as an embedded Scala DSL. It generates an intermediate representation called FIRRTL [15], which can in turn be compiled into Verilog. Like Verilog, Chisel can describe hardware at the RTL level, which makes it distinct from high-level synthesis languages such as SystemC. However, unlike Verilog, Chisel can use the power of the Scala language to build complex RTL generators, thus reducing the repetitiveness usually associated with RTL design. To give an example of how powerful Chisel's generator capabilities are, let's take a look at implementing a generator for a parallel scan circuit.

A scan operation involves taking an input vector X of degree n and applying a binary associative operator $\bigoplus$ to each element so that you produce an output vector Y of degree n

defined as follows:

$$Y_i = \left\{ \begin{array}{ll} X_i, & \text{for } i = 0 \\ X_i \bigoplus Y_{i-1}, & \text{for } 1 \leq i < n \end{array} \right\} \tag{4.1}$$

The most common variation of scan is a prefix sum, which uses addition as the binary operator. In this case, each element Y_i would simply be $\sum_{j=0}^{i} X_j$. This is fairly simple to compute sequentially on a CPU, but the naive approach does not translate well to a hardware implementation, as it would result in an $O(N)$ long chain of adders as shown in Figure 4.1a. Instead, hardware implementations use parallel scan algorithms like the Hill-Steele algorithm (shown in Figure 4.1b), which can implement scan in $O(log_2(N))$ stages.



(a) Naive

(b) Hill-Steele

Figure 4.1: Prefix Sum

An implementation of prefix sum using the Hill-Steele algorithm in Chisel is shown in Figure 4.2. This is not just an RTL description of a single circuit. It is a generator for a class of circuits with a configurable number of integer inputs of configurable width. Describing hardware using generators in this way reduces the repetitiveness of RTL coding. If the complete design needs multiple prefix sum units of different sizes, the same generator can be used for all of them instead of writing new code for each circuit. Being able to change parameters in the circuit also helps with design-space exploration, as the designer can test multiple configurations of the circuit without writing new code for each.

The level of configurability shown in Figure 4.2 can already be achieved in existing languages like SystemVerilog using parameters and generate statements. The true strengths

```
class PrefixSum(n: Int, w: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(Vec(n, UInt(w.W)))
    val out = Output(Vec(n, UInt(w.W)))
  })

  val nStages = log2Ceil(n)
  val stages = Wire(Vec(nStages, Vec(n, UInt(w.W))))

  for (i <- 0 until nStages) {
    val lastStage = if (i == 0) io.in else stages(i-1)
    val off = 1 << i

    for (j <- 0 until n) {
      if (j < off)
        stages(i)(j) := lastStage(j)
      else
        stages(i)(j) := lastStage(j-off) + lastStage(j)
    }
  }

  io.out := stages.last
}
```

Figure 4.2: Chisel implementation of Prefix Sum

of Chisel over other HDLs can be seen by implementing a generic parallel scan generator, as shown in Figure 4.3. This generator can not only configure the number and width of inputs, but also the binary operator. Figure 4.4 shows two different ways of implementing an 8-element prefix sum using 32-bit integers. The first instantiation uses the PrefixSum generator from Figure 4.2, and the second uses the more general Scan generator from Figure 4.3. To make the Scan module generate prefix sums, the addition operator is passed in as an anonymous function. Of course, other operators can be used as well. For instance, by changing addition to multiplication, we can implement a prefix product circuit instead, as shown by the definition of prod0 in Figure 4.6. The use of a first-class function allows us to reuse the same generator framework to produce any arbitrary scan circuit with minimal additional code, something that would be difficult to achieve in SystemVerilog. But by far the most powerful feature of Chisel is its ability to leverage the Scala type system. Note that the definition of the Scan module allows other data types besides a simple unsigned integer. For instance, if we implement a complex signed integer datatype with an overridden

multiplication operator (Figure 4.5), we can use that in the invocation of Scan and produce a prefix product over complex numbers rather than real integers (prod1 in 4.6).

```
class Scan[T <: Data](n: Int, typ: T, oper: (T, T) => T) extends Module {
  val io = IO(new Bundle {
    val in = Input(Vec(n, typ.cloneType))
    val out = Output(Vec(n, typ.cloneType))
  })

  val nStages = log2Ceil(n)
  val stages = Wire(Vec(nStages, Vec(n, typ.cloneType)))

  for (i <- 0 until nStages) {
    val lastStage = if (i == 0) io.in else stages(i-1)
    val off = 1 << i

    for (j <- 0 until n) {
      if (j < off)
        stages(i)(j) := lastStage(j)
      else
        stages(i)(j) := oper(lastStage(j-off), lastStage(j))
    }
  }

  io.out := stages.last
}
```

Figure 4.3: Chisel implementation of Scan

```
    val sum0 = Module(new PrefixSum(8, 32))
    val sum1 = Module(new Scan(8, UInt(32.W),
        (a: UInt, b: UInt) => (a + b)))
```

Figure 4.4: Two different ways of implementing prefix sum

```
class SIntComplex(val w: Int) extends Bundle {
  val real = SInt(w.W)
  val imag = SInt(w.W)

  def *(that: SIntComplex): SIntComplex = {
    val prod = Wire(new SIntComplex(w))
    prod.real := this.real * that.real - this.imag * that.imag
    prod.imag := this.imag * that.real + this.real * that.imag
    prod
  }
}
```

Figure 4.5: Complex Signed Integer

```
    val prod0 = Module(new Scan(8, UInt(32.W),
        (a: UInt, b: UInt) => (a * b)))
    val prod1 = Module(new Scan(8, new SIntComplex(32),
        (a: SIntComplex, b: SIntComplex) => (a * b)))
```

Figure 4.6: Prefix products using Scan generator

4.2 Chisel Libraries

The ability to write highly configurable hardware generators makes Chisel well-suited for developing reusable hardware libraries. In our lab, we maintain a growing ecosystem of hardware libraries that make developing and evaluating new hardware designs more productive. The following are some of the libraries we leverage to build our remote memory designs.

Rocket Chip

The core of our library ecosystem is the Rocket Chip SoC generator [3]. This is a collection of SoC components centered around the Rocket CPU. Rocket is an in-order 64-bit RISC-V application core with support for all the standard RISC-V extensions, such as integer multiply and divide, single and double-precision floating-point operations, compressed instructions, and virtual memory. As shown in Figure 4.7, SoCs generated using Rocket Chip contain one or more Rocket CPUs attached to an on-chip network which includes a coherent multi-level memory system with separate L1 instruction and data caches, shared L2 cache banks, and a DRAM controller using the AXI4 memory interface. The on-chip network also

includes memory-mapped IO devices like a BootROM for storing the first-stage bootloader, a platform-level interrupt controller (PLIC) for managing external interrupts, a core-local interrupt (CLINT) module for software and timer interrupts, and a debug unit for accessing memory and issuing instructions through JTAG.

Rocket Chip has an advanced configuration system based on context-dependent environments [8], which allows the generator to produce different SoCs simply by creating different configuration objects. Figure 4.8 shows how configuration fragments can be layered onto a base configuration to produce two radically different design points. The first configuration (BigConfig) contains four RV64 application cores with floating-point units and virtual memory. The memory system contains 16 kB L1 instruction and data caches backed by a shared 4-bank 512 kB L2 cache and DRAM main memory. By contrast, the LittleConfig has a small RV32 core with no virtual memory or floating point support. It has a L1 data scratchpad instead of a data cache and no backing memory. The former configuration is similar to what one might find in the application core of a mobile phone, while the second is closer to a low-power microcontroller. Yet it only took two additional lines of code to create each configuration from the base configuration.

The various components of Rocket Chip are connected together through the TileLink cache-coherence/memory protocol [13]. Rocket Chip's implementation of TileLink allows the SoC to be easily extended with additional peripherals and DMA devices, a feature we rely on extensively for other libraries in the ecosystem. As shown in Figure 4.7, MMIO peripherals are connected to the periphery bus, thus allowing the CPUs to read and write from their MMIO registers. DMA devices connect to the frontend bus, allowing them to read and write into the coherent memory system.

Hwacha

While Rocket is useful for basic application core tasks like hosting an OS kernel, its raw compute performance is limited. In order to perform high bandwidth data-parallel computations, we use the Hwacha [22] vector coprocessor, shown in Figure 4.9. Hwacha takes custom instructions from the application core through the Rocket Custom Coprocessor (RoCC) interface. However, Hwacha uses a decoupled vector-fetch architecture, so not all of the instructions come directly from the application core. The instructions sent through RoCC only set the scalar and address registers, the vector length, and configuration register. To perform actual vector compute and memory operations, the application core sends a vector-fetch instruction, which causes the coprocessor's internal instruction cache to fetch and issue vector instructions from a separate instruction stream. The vector instructions are issued to a master sequencer, which breaks each instruction into "strips", each of which has eight 64-bit elements. The strips are then distributed across multiple vector lanes, each of which contains a vector register file, predicate register file, functional units, and vector memory units. The functional units take data from the vector registers, perform the specified operations, and then write them back to the vector registers. The VMU takes data from the vector registers and stores it to L2 or reads data from the L2 and write it to the vector

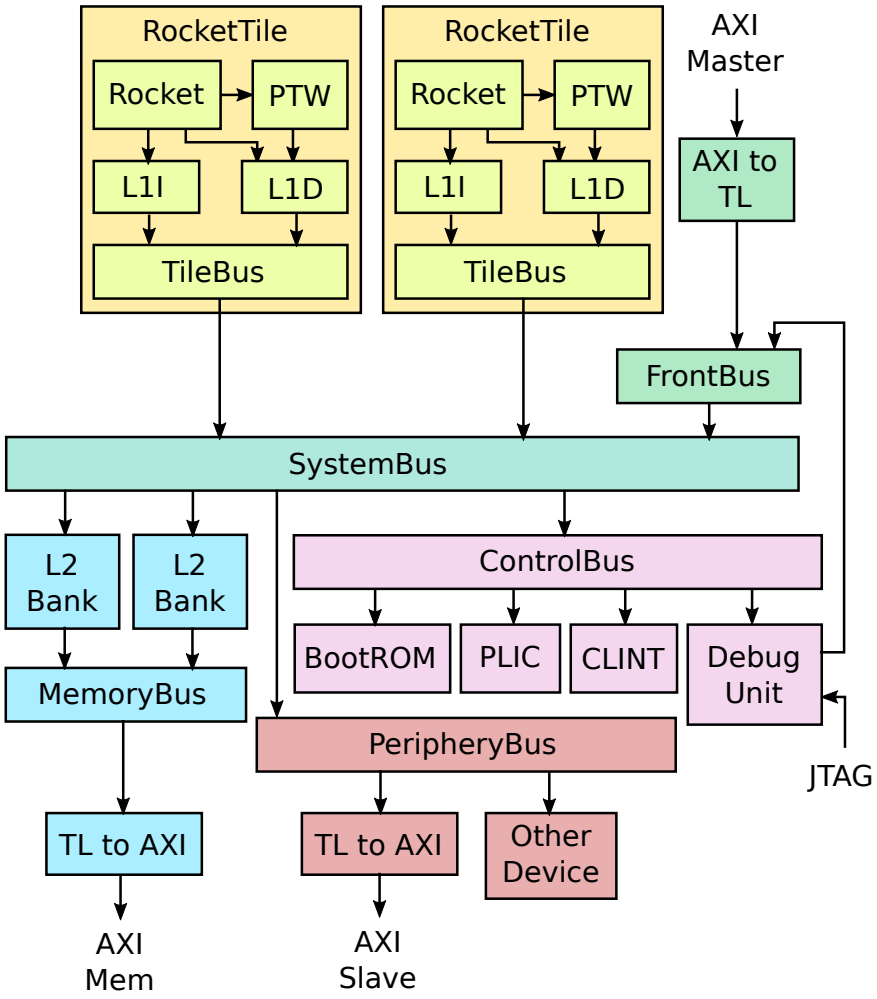


Figure 4.7: A Basic Rocket Chip SoC

```
class BigConfig extends Config(
    new WithNBigCores(4) ++
    new WithInclusiveCache(nBanks = 4) ++
    new BaseConfig)

class LittleConfig extends Config(
    new With1Tinycore ++
    new WithNoMemPort ++
    new BaseConfig)
```

Figure 4.8: Big and Little Rocket Chip Configurations

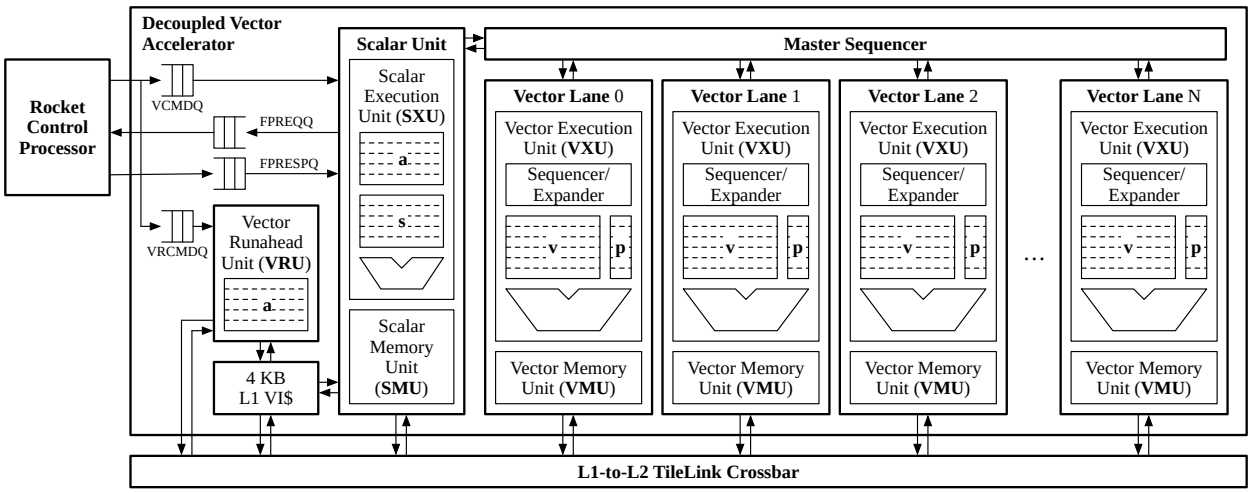


Figure 4.9: Hwacha [22]

registers. The VMU also performs address translation, allowing Hwacha instructions to use virtual addresses. The scalar registers are write-only from the application core and read-only from the vector lanes. The contents of the scalar registers are copied to the master sequencer on a vector-fetch instruction. Therefore, executing scalar instructions will not perturb the state seen by the vector lanes, nor does the scalar unit depend on any information from the vector lanes. This allows the application core to run ahead of the coprocessor and queue up vector fetch blocks, thus leading to efficient pipelining of work.

Boom

Hwacha is useful for accelerating data parallel, bandwidth-sensitive workloads, but it cannot accelerate serialized, latency-sensitive operations, such as an OS page-fault handler. For

those types of tasks, we instead use the BOOM out-of-order core [6]. BOOM is a superscalar out-of-order core with explicit register renaming. Like Rocket, it implements all the standard RISC-V ISA extensions as well as the RoCC coprocessor interface. Also like Rocket, BOOM is a generator and not just an implementation of a single design point. Microarchitectural parameters like the issue width, the number of physical registers, the size of the ROB, the branch predictor, and even the number of functional units can all be configured.

IceNet

IceNet is a Chisel library that provides an on-chip Ethernet network interface controller (NIC) that can hook into the Rocket Chip generator. As shown in Figure 4.11 the NIC has four main parts: a send path for reading packets from memory and sending it out to the network, a receive path for taking packets from the network and writing to memory, a controller that interfaces between the CPU and the send and receive paths, and a pause handler that handles flow control.

The controller interfaces with the CPU by exposing four queues as memory-mapped IO registers: the send request queue, the send completion queue, the receive request queue, and the receive completion queue. The request queues are write-only, while the completion queues are read-only. To send a packet, the CPU writes a request containing the starting address and length (in bytes) of the packet to the send request queue. The request is then forwarded to the send path, which sends a completion back to the controller once it is finished processing the request. This completion goes into the send completion queue, which sends an interrupt to the CPU. The send interrupt handler in the NIC kernel driver then releases any resources associated with the request and reads from the send completion queue to clear the entry. For receiving packets, the kernel driver allocates buffers sized to hold the largest possible Ethernet packet. It then writes the addresses of these buffers into the receive request queue. When the receive path gets a packet from the network, it pops an entry off the receive request queue, writes the data into that buffer, and sends a receive completion containing the number of bytes in the packet back to the controller. The completion goes into the receive completion queue, which sends an interrupt to the CPU. The receive interrupt handler in the kernel driver then reads the completion off the queue, delivers the data in the buffer to the userspace program waiting for it, and pushes a new buffer onto the request queue.

The send path is itself made of several components. The first is the reader, which takes the send requests from the controller and sends out TileLink read requests to the memory system. The data returned from memory can come out of order, so it goes into a reservation buffer, which stores them in the correct order. The data then comes out of the reservation buffer ready to send, but it might share the network interface with one or more input taps. These taps come from other devices that wish to access the network, such as our memory blade controller, remote memory client controller, and DRAM cache controller, which we will discuss in Chapters 6 and 7. If there are input taps, an arbiter is used to arbitrate between them and the reservation buffer. The arbiter switches between the various inputs packet by packet in a round-robin fashion.

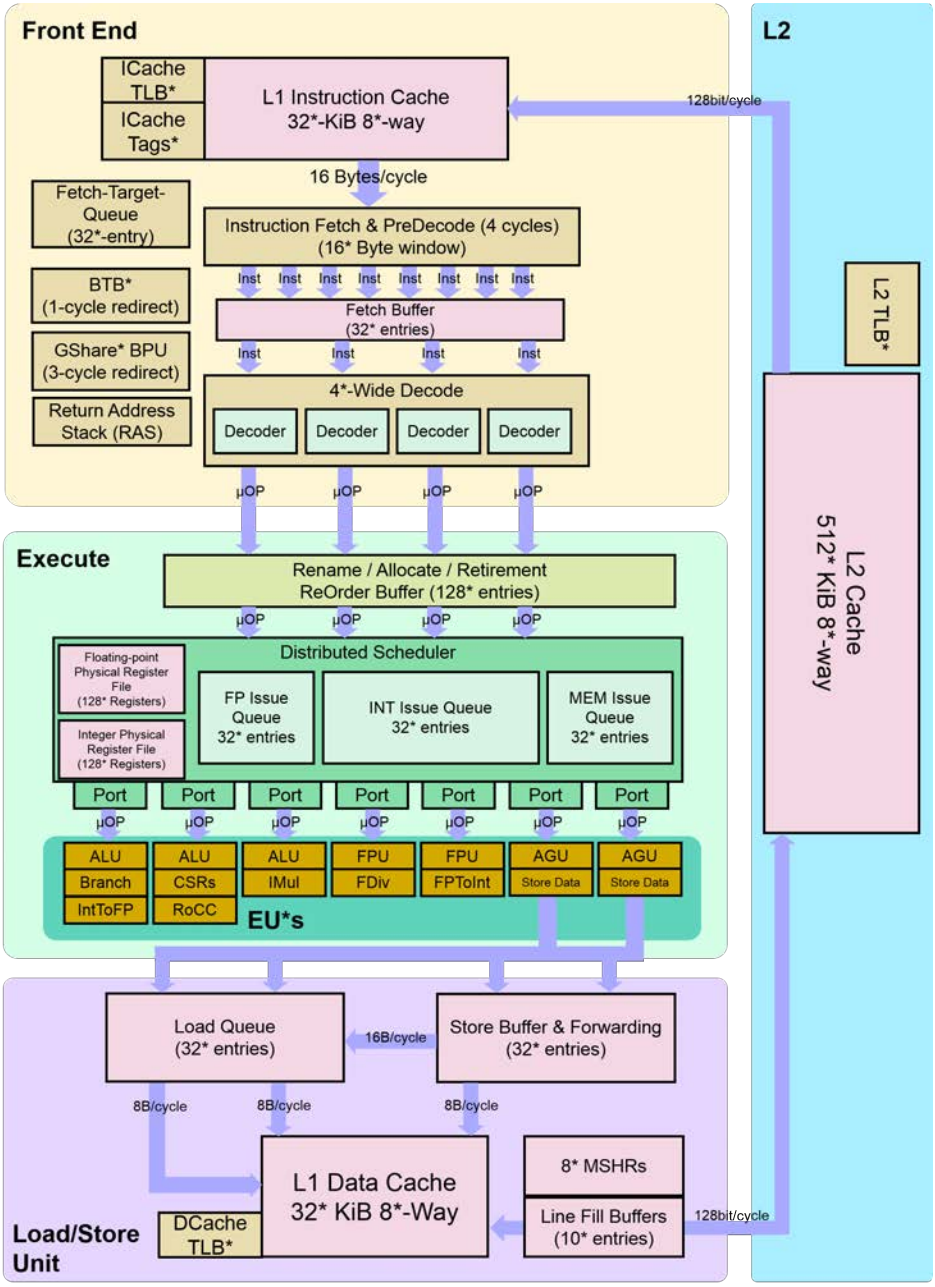


Figure 4.10: BOOM

The receive path has two main components: the packet buffer and the writer. When a packet arrives to the NIC from the network, it first goes into the packet buffer, which stores the data in SRAM. If there isn't enough free space left when a packet arrives, the packet buffer will make sure that data is dropped at packet boundaries so that the kernel driver will not see an incomplete packet. Data then comes out of the packet buffer and goes to the writer, which takes an entry off the receive request queue and writes the data to the address given in the receive request before sending a receive completion back to the controller. The receive path can be configured to have output taps. These taps go to other devices which wish to receive network packets. Generally, these will be the same devices that use the input tap on the send path. These taps are implemented by adding an additional packet buffer for each tap. The data coming from the network will be duplicated into each buffer, but the packet buffer connected to the writer will drop any packets meant to be tapped out, and the packet buffer for each tap will drop any packets not meant for that particular tap.

The pause controller sits between the send/receive paths and the network interface. Its job is to handle flow control using Ethernet pause frames. The pause handler tracks the occupancy of the packet buffer in the receive path. If the occupancy of the buffer exceeds a given threshold, the handler will send an Ethernet pause frame, which tells the switch on the other side of the Ethernet link to stop sending packets for a given period of time. During this time, the packet buffer gets a chance to drain. If the buffer occupancy is still over the threshold when the pause is due to expire, the handler will send another pause packet. This will continue until the buffer drains sufficiently. If the pause handler receives a pause frame from the network, it will halt the sending of data from the send path.

4.3 Chapter Summary

In this chapter, we have seen the various tools developed in our lab to make RTL development more productive. Chisel provides a generator API that allows RTL designers to write code more efficiently and reusably, Rocket Chip and other libraries provide a ready-made collection of common components, TileLink and Diplomacy provide a framework for adding custom peripherals, and the Rocket Chip configuration system provides a way of quickly spinning out multiple design points for design space exploration. However, simply producing RTL code is only one part of RTL development. Designers also need a way to simulate, debug, and evaluate their design. Figure 4.12 shows how the various tools come together to form a complete tool flow, which can take a configuration and produce either a software simulation model or an FPGA-based simulation model. In the next chapter, we will discuss these simulation models and our custom tools for fast, accurate RTL simulation and debugging.

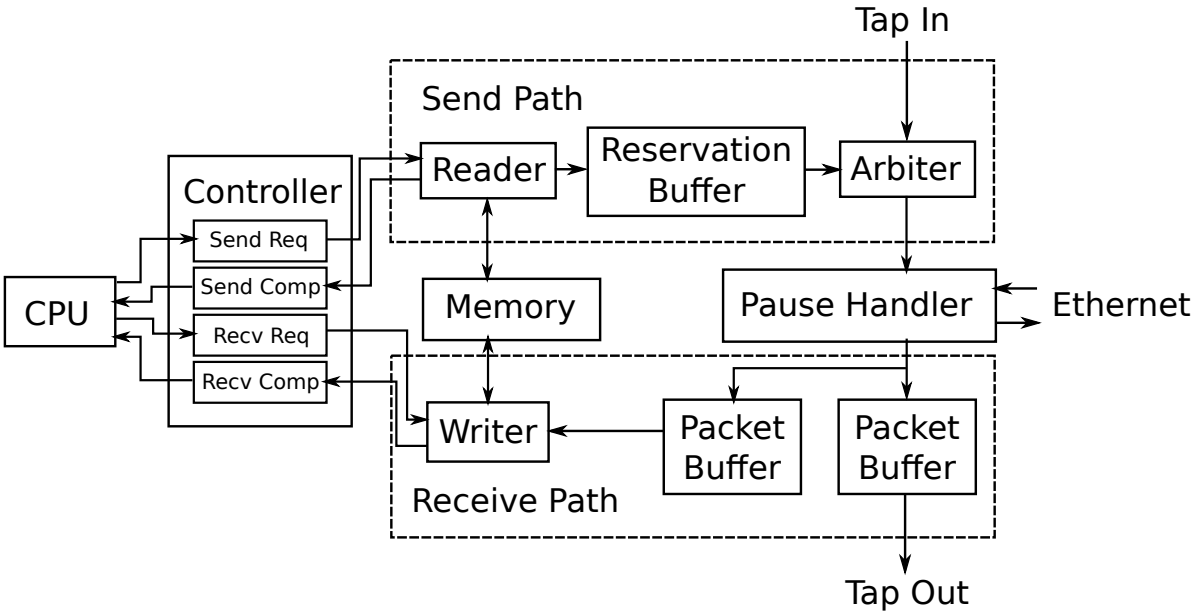


Figure 4.11: IceNet NIC

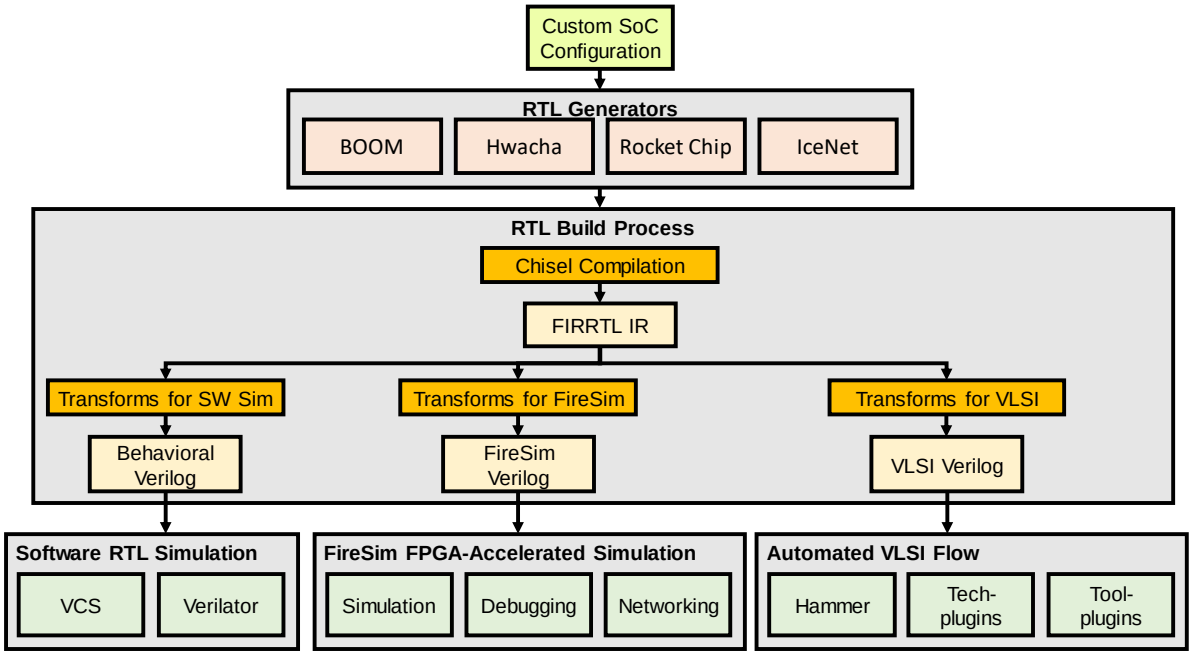


Figure 4.12: Complete Tool Flow

Chapter 5

Simulation Platform

Chisel and the various hardware libraries based on it make the task of developing RTL much easier. However, once the RTL is written, it needs to be simulated to verify the design and measure performance. Commercial or open-source software simulators like VCS and Verilator can be used to simulate designs during development for verification purposes. However, they are too slow to run longer workloads, such as booting the Linux Kernel. Software simulations also cannot easily simulate multi-node, networked systems, such as a data center rack with both compute nodes and memory blades. For these sorts of large-scale simulations, designers generally use field-programmable gate arrays (FPGAs), which can simulate large designs at much faster rates than is possible with software simulators. However, one issue of using FPGAs directly is that the simulation timing is dependent on the timing of the hardened FPGA resources (such as DRAM and network). This can be an issue if, for instance, the target clock rate of the circuit being simulated is higher than what the FPGA can achieve. For instance, consider a chip designed to be taped out at 1 GHz and connected to a DRAM with a 100 ns average latency (Fig. 5.1). The same RTL mapped to an FPGA obviously cannot achieve 1 GHz. More realistically, the FPGA frequency would be around 100 MHz. However, if the DRAM on the FPGA is the same as the DRAM that would eventually be attached to the taped-out chip, its latency would appear 10x faster in terms of clock cycles. This makes performing accurate performance validation/benchmarking on the FPGA difficult. The obvious strategy of running a workload on the FPGA and then scaling the result by 10x would make the performance seem unrealistically fast, especially if it is a memory-bound application.

To get accurate performance results from the FPGA, we must create a DRAM timing model instead of using the on-board DRAM directly. This timing model would intercept memory requests from the RTL model, add the needed delay to it, and then forward the request to the memory. If the request latency is fixed, creating the model is relatively straightforward, a simple latency pipe would do, as in Figure 5.2. Just adding on the additional 90 cycles of latency will make the overall latency of the memory appear to be the same number cycles as it would in the taped out chip.

But of course, reality is rarely so simple. DRAM latency is not fixed, but rather varies

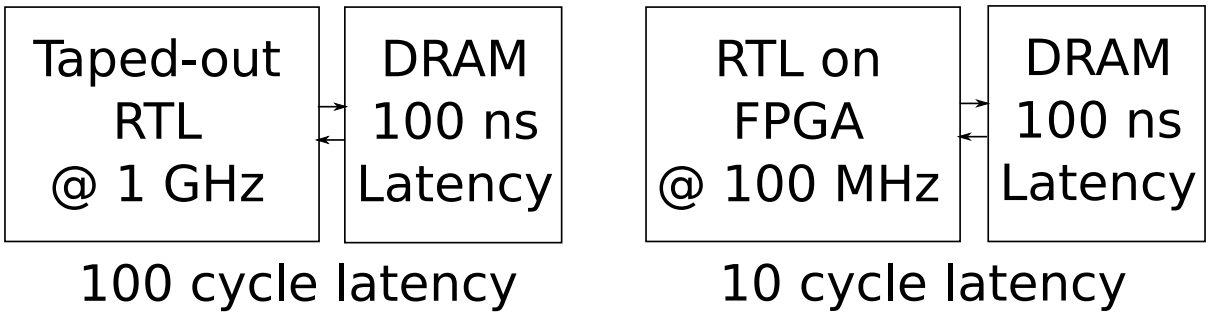


Figure 5.1: Comparison of Taped-out RTL (left) to its simulation on FPGA (right)

based on factors such as row-buffer locality. Accurately modeling that would require a more sophisticated model that both delivers varying latency to the target and accounts for varying latency of the host DRAM. Another issue is if the target latency a model wants to achieve is actually lower than the latency the host resource can provide. In that case, the target clock must be paused, which means the target RTL must be modified from what would eventually be taped out. Doing this manually could be tedious and error-prone, which is why our lab has developed tools that can generate such models from the target RTL automatically.

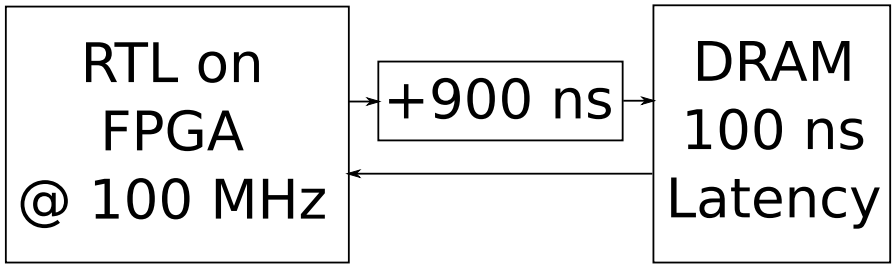


Figure 5.2: Latency Pipe Model

5.1 Golden Gate

The core of our simulation platform is the Golden Gate compiler. It takes as input the FIRRTL intermediate representation generated by the Chisel compiler for the target RTL and performs code transformations on it to produce an FPGA model. This model decouples the target clock from the host clock through clock gating. The transformed RTL is then connected to an FPGA DRAM model (and other FPGA models) through two token queues, one in each direction. On every target cycle, each model consumes a token from its input

and produces a token on its output. If there is no input token available on a host cycle, the target clock will not advance on that host cycle. In this way, the various FPGA models can keep their target times in sync with each other, even if the host timings differ.

Consider our previous example of a CPU SoC connected to a single DRAM channel. With Golden Gate, the SoC RTL gets transformed into a host-decoupled FPGA model and connects to a DRAM model through a request queue and a response queue (Fig. 5.3). The tokens sent on these queues serve both to carry the request/response information and to keep track of timing. The tokens have a valid bit to indicate whether or not they are actually carrying data. The DRAM model is hosted on the FPGA and keeps track of the metadata it needs to properly model the timing of a DRAM device. It communicates with the host DRAM to store and retrieve the actual data. Upon receiving a valid token from the request queue, the model calculates how many target cycles are necessary to fulfill the request and puts an equivalent number of null tokens on the response queue. Meanwhile, it forwards the request to the DRAM. When the response comes back from DRAM, it puts the response in the queue after the null tokens. If the null tokens run out before the host DRAM responds, the CPU target clock will stop advancing. This way, the target timing is precise and deterministic. If the same workload was run again, the target DRAM latency would be exactly the same, even if the latency of the host DRAM were different.

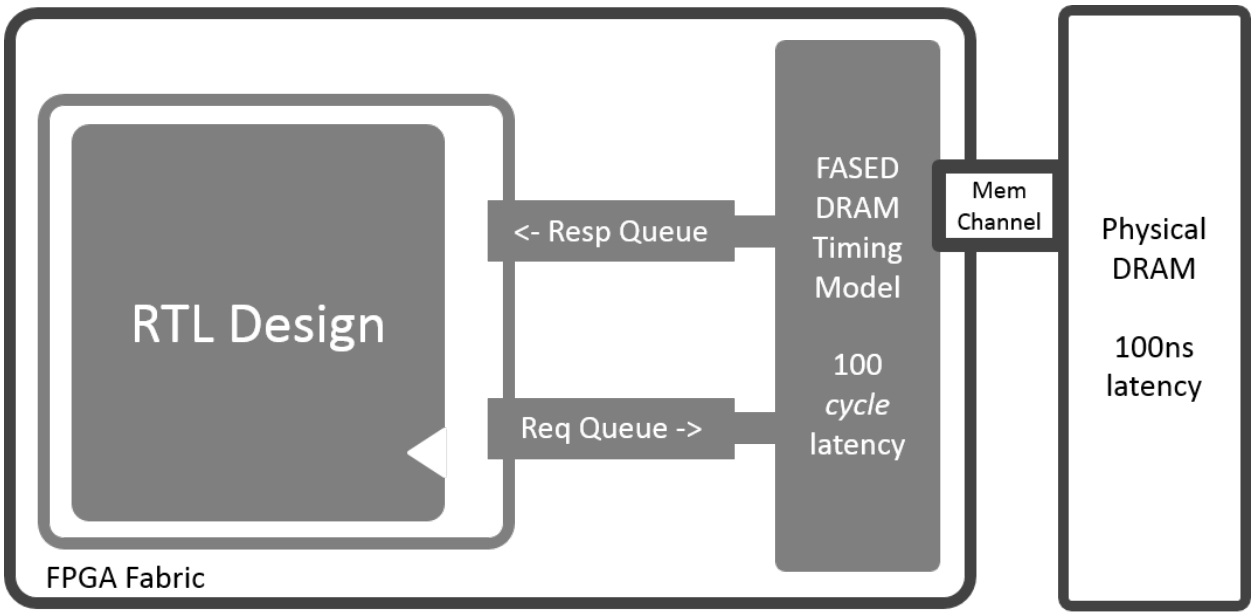


Figure 5.3: Golden Gate DRAM Model

One benefit of tracking target time by passing tokens is that it is a very flexible model. The tokens are agnostic to the medium through which they are transported and need not stay in FPGA registers and SRAM. They can be written to DRAM, carried over PCIe, or otherwise transferred to the host CPU for software processing. For instance, if we wished to

model SSD rather than DRAM, we could send the tokens to the CPU, which can compute the timings in software and source the data from the host's tertiary storage 5.4. We use this flexibility to develop the various models used for FireSim, our cloud-based FPGA-accelerated simulation platform.

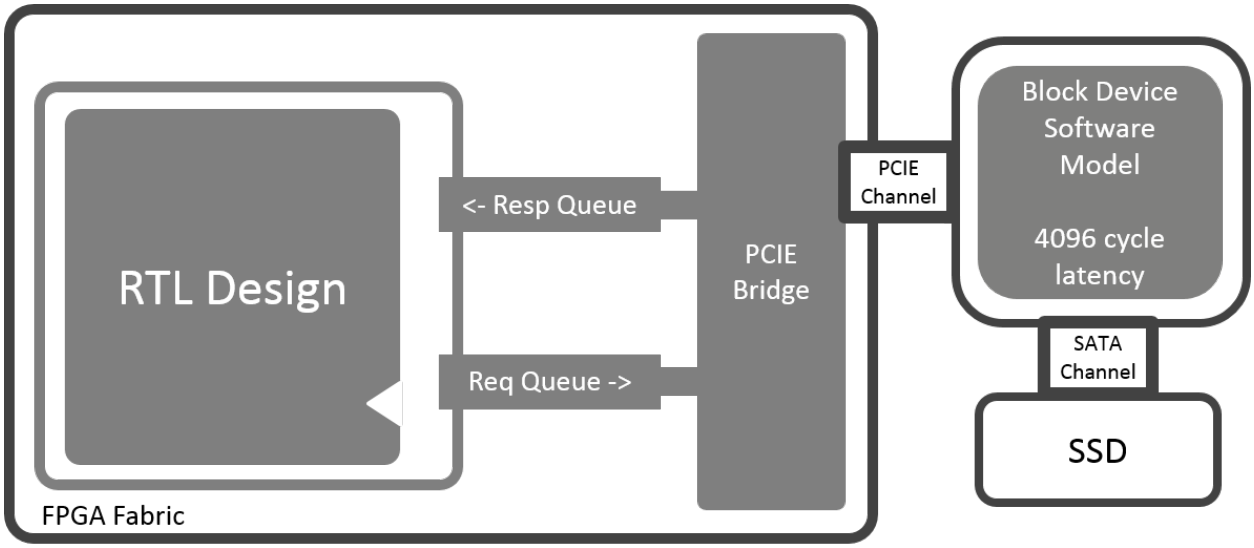


Figure 5.4: Golden Gate SSD Model

5.2 FireSim

FireSim [18] is a cloud-based, FPGA-accelerated simulation platform built on top of the Rocket Chip ecosystem and Golden Gate compiler. It leverages Amazon Web Services' f1 instances, which can come with 1, 2, or 8 PCIe-attached Xilinx FPGAs. AWS provides a shell for accessing the FPGA-attached DRAM and connecting to the PCIe, as well as an SDK for interacting with the FPGA through PCIe from the CPU. We use these tools to connect the FPGA-hosted DRAM and block device models described in the previous section. We also used the AWS API to create a FireSim manager tool, which can automatically spin up f1 instances and deploy FPGA images and workloads to them.

Network Simulation

The key feature of FireSim is its ability to simulate networked clusters across multiple FPGAs. The standard SoC modeled by FireSim includes the IceNet NIC described in Chapter 4. The network flits going in and out of the NIC are transformed by Golden Gate into timing tokens, which are carried between the FPGA and host CPU via PCIe. On the CPU side, the tokens go into a software switch model, which implements store-and-forward routing of

network packets. On ingress, valid tokens are buffered and grouped into packets, given a timestamp based on the arrival time of the last flit, and stored in input queues (one for each input interface). Then, in a global routing step, the packets in all of the input queues are drained, sorted by timestamp, and distributed to output queues based on a fixed MAC address table. The packets are then delivered from the output queues to the FPGAs once bandwidth is available, all Ethernet pauses have expired, and the target time of the simulation has advanced past the delivery timestamp of the packet. This keeps the simulation synchronized and cycle-accurate across all FPGAs.

The switch model can not only deliver tokens between multiple FPGAs on an instance, they can also tunnel tokens through TCP and thus carry them across multiple instances. Transferring packets between simulations on different instances requires an intermediate switch model on a third instance acting as a top-of-rack switch. The TCP-based ports between instances are treated the same as the PCIe-based ports to the FPGAs as far as routing and timing goes. Figure 5.5 shows how the various models are mapped to the FPGAs and CPUs on the EC2 instances.

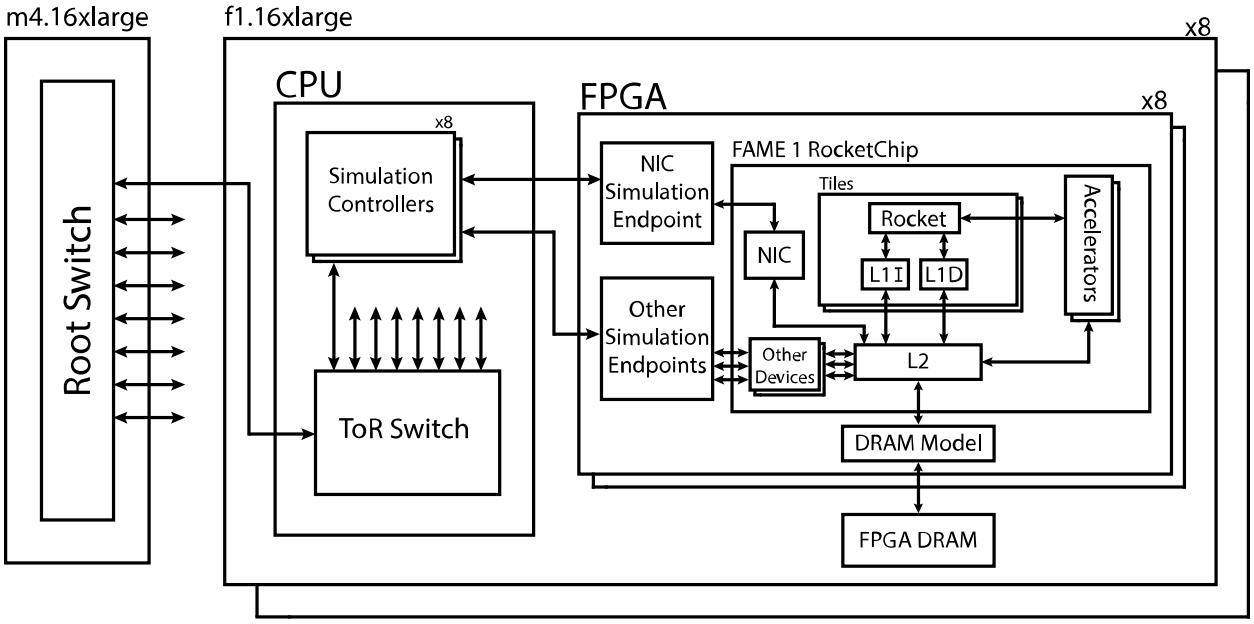


Figure 5.5: Mapping simulation to EC2 F1 in FireSim [18]

Management Tools

Besides the FPGA and software simulation models, FireSim also provides software management tools to automate tasks like building FPGA images, building target software images, and spinning up simulations. The FireSim manager handles the first two tasks using the

AWS API. To build FPGA images, the manager runs the Chisel and Golden Gate compilers to produce Verilog for the simulation model, spins up a standard EC2 instance (i.e., without attached FPGAs), runs Vivado on the new instance to build an FPGA image, and then produces an Amazon FPGA image (AFI) from the result. To run simulations, the manager spins up the required number of f1 and m4 instances (the m4 instances are used for the top-of-rack switches). It then copies over the simulation executables for interfacing with the FPGA, the executables for the switch models, the RISC-V executables for the target software, and the disk images for the block device models. It then flashes the FPGAs with the pre-built AFI, sets up shared memory files for the simulation shells to communicate with the switch models, and runs all of the host executables with their outputs logged to files. It then waits for any of the simulation processes to exit, at which point it kills the remaining processes and copies the output logs and disk images back to the manager instance. Finally, the manager mounts the disk images that were copied back and extracts any specified target output files from them.

The construction of target software images is handled by the FireMarshal tool. FireMarshal builds two things: the executable Linux image and the disk image. The tool is given a configuration file in JSON format that specifies how to configure the kernel and filesystem. For the Linux image, FireMarshal pulls in three different pieces: kernel driver modules for the NIC, block device, and other peripherals; the RISC-V kernel itself; and the Berkeley Bootloader (BBL). The kernel modules are built and then placed in an initramfs. The kernel is then built with the module initramfs included in it. Finally, BBL, a machine-mode bootloader for RISC-V systems, is built with the kernel image as a payload. When BBL is run in simulation, it unpacks the kernel image to memory and executes it. The kernel then loads the kernel modules from the initramfs, mounts the disk image, and executes init from it. The disk image is based on Buildroot, a Linux distribution for embedded systems. FireMarshal uses the Buildroot tool to generate an EXT2 filesystem image, which includes packages configured through Buildroot and files from an overlay directory specified in the JSON configuration. The JSON configuration can also specify a command to be run automatically when the image boots, which FireMarshal adds to an init script in the overlay.

Profiling and Debugging

Using the Golden Gate compiler, FireSim is able to simulate large RTL designs at a much faster rate than software RTL simulators while also providing greater cycle accuracy than an abstract software simulator or FPGA prototype. However, speed and accuracy of simulation are not the only factors hardware designers look for when choosing a simulation platform. Also important is the degree of introspection that the simulator offers into the design. Being able to see the architectural and microarchitectural state of the design as it executes a workload is invaluable to debugging and performance profiling.

Two key features of software RTL simulators used for debugging hardware are print and assert statements. Print statements allow the designer to output the values of hardware signals. Assert statements cause the simulation to abort if certain conditions are violated.

These two features make it much easier to detect anomalies and track down the root causes of bugs. With an ordinary FPGA prototype, it wouldn't be possible to implement such features, as they are complex operations that would necessarily require multiple cycles to perform. However, since Golden Gate FPGA models have target decoupling, it is possible to implement assertions and prints by pausing target time as information is transferred from the FPGA to the host CPU. Golden Gate implements prints and asserts by assigning each assertion and annotated print statement a number. When an assertion or a print statement fires, the target is paused and the number of the assertion or print statement is transmitted to the host. The host simulation driver then looks up the appropriate message in a table and prints it out. For print statements, the FPGA also transmits the values of the hardware signals given as arguments to the print statements.

A useful feature for developing and profiling CPU hardware specifically is the ability to produce a trace of the instructions the CPU executes. This is a standard feature of abstract architectural simulators. In FireSim, we accomplish this using the TracerV utility. Each CPU in the SoC has a trace port which outputs the instruction at the end of its pipeline (or, for OoO CPUs, the instruction at the head of the reorder buffer) on every cycle, along with a bit to indicate whether or not the instruction is being committed. The Golden Gate FPGA wrapper stores this data in an on-chip buffer. When the buffer is full, the simulation is paused as the CPU reads the trace data and outputs it to a file. Once the data has been read, the buffer can be cleared and the simulation resumed.

For a long-running workload, such as a program running in RISC-V Linux, an instruction trace starting from the beginning of simulation is neither necessary nor desirable. The user is interested in the instructions executed once the userspace program begins, but to reach that point, the simulation must first run a long kernel boot sequence. Tracing instructions during this time needlessly slows down simulation due to the frequent PCIe transfers and quickly fills up the disk with data that will likely be ignored later. Therefore, TracerV provides three different types of triggers to start and stop collecting trace data. The first is a cycle count trigger, in which tracing starts when the target reaches a start cycle and ends when it reaches an end cycle. The second is a PC-based trigger, which matches on the instruction address being executed by the CPU. The third and final trigger is an instruction trigger, which matches on the instruction data itself rather than the instruction address. Of these, the instruction data trigger is the one we use most often. Generally, we trigger on NOP instructions added to the userspace program. The RISC-V ISA does not have a dedicated NOP instruction, but they can be constructed with regular integer instructions that set the x0 register as their destination register. This register always has a value of zero and ignores any writes to it. Since the source registers and immediates can be varied without changing the result, this gives us a large number of NOP instructions to match on. Figure 5.6 provides an example of how to insert triggering instructions into a C program.

The TracerV trace contains not only the instructions that executed, but also the precise cycles at which they executed. This means that the information provided by the trace can be used to construct a very accurate profile of the workload. FireSim includes a set of tools called FirePerf [19] for performing this task. Using the trace and DWARF debugging

```
asm volatile ("add x0, x1, x1"); // Start trigger
dowork();
asm volatile ("add x0, x2, x2"); // End trigger
```

Figure 5.6: Example start and end trigger instructions

information from the vmlinux image, FirePerf can determine exactly how long the kernel stays in each function. It then creates a flame graph displaying this information (example in Fig. 5.7). Each bar in the flame graph represents a distinct function and the length of the bar represents how much time the CPU spent within that function. Function bars stack on top of each other to show the stack of subroutine calls. This gives the user an intuitive visual representation that makes it straightforward to determine which functions to target for optimization. This tool is similar to software profilers like perf and gprof, but delivers far more accurate results for two reasons. First, TracerV collects the complete execution history of the program at single cycle resolution, whereas software profilers merely sample the instruction stream periodically. Second, TracerV collects data without changing target timing, whereas software profilers necessarily disturb the target execution by periodically interrupting it to collect samples.

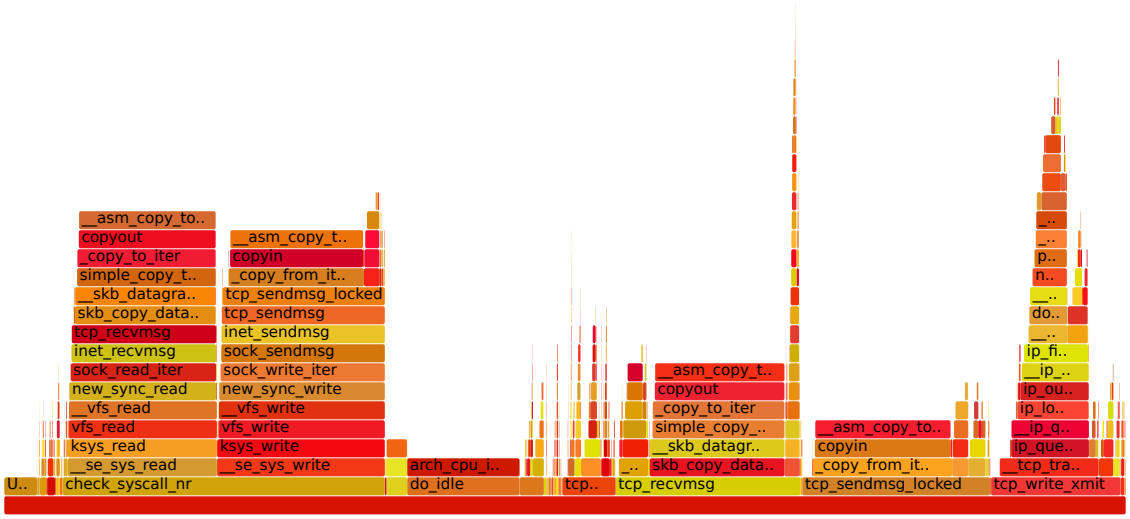


Figure 5.7: Example FirePerf Flame Graph [19]

5.3 Chapter Summary

In this chapter, we've discussed the simulation and debugging tools provided by the FireSim suite. FireSim provides a framework for combining RTL and software models to produce a fast, cycle-accurate, multi-node simulation. Through the use of FAME transforms, we can produce an FPGA model that maintains target timing relative to the hardened FPGA resources without manually changing the RTL. The cycle-accurate network model allows FireSim to simulate an entire cluster of servers and not just a single node. In the next chapter, we will use FireSim to evaluate our baseline design: a remote memory system with dedicated memory blades and client nodes equipped with RDMA controllers.

Chapter 6

Memory Blade and Remote Memory Client Implementation

Using Chisel, Rocket Chip, and FireSim, we can begin implementing and simulating the hardware we will need for our disaggregated memory data center rack. Before we design and test the DRAM cache, we would first like to build a system similar to existing RDMA hardware. This will serve as a baseline to compare later results against. We will still have separate memory blades and compute blades, but instead of using a DRAM cache, the compute blades will have an RDMA client controller to explicitly move data between its local DRAM and the remote memory. We will first take a look at the remote memory protocol that the client and memory blade use to communicate with each other. Then we will look at the microarchitecture of the remote memory client and memory blade. Finally, we will show the results of two benchmarks: one that simply measures the raw bandwidth and latency and one that shows how much latency is added by mediating remote memory access through the virtual memory system.

6.1 Remote Memory Protocol

The compute blades and memory blades communicate with each other through a remote memory protocol, which is carried over Ethernet. Each packet starts with an Ethernet header, which uses custom EtherTypes to distinguish remote memory requests and responses from other packets. We use the EtherType 0804 for requests and 0805 for responses. For remote memory requests, the Ethernet payload starts with a remote memory request header, as shown in Table 6.1. This header includes an opcode to indicate which type of request it is, a span ID to indicate which address on the memory blade is being accessed, and a unique transaction ID to allow multiple in-flight requests to be distinguished from each other.

To keep the hardware simple, we split the memory blade's address space into relatively large fixed-size spans. The size of a span will be configured at hardware compile time. For the experiments in this chapter, the span size will be set to 1024 bytes, which is the largest

Field	Octets	Values	Description
version	1	0x02: current draft	Protocol version
opcode	1	0x00: span read 0x01: span write	Operation identifier
reserved	2		
xact_id	4		Unique transaction identifier
span_id	8		Span ID to read

Table 6.1: Protocol request header

Field	Octets	Values	Description
version	1	0x02: current draft	Protocol version
resp_code	1	0x80: success, w/ span data 0x81: success, no data $\geq$ 0x83: failure	Operation status
reserved	2		
xact_id	4		Unique transaction identifier

Table 6.2: Protocol response header

power-of-two size that can fit into a standard Ethernet frame (maximum transmission unit is 1500 bytes). Our protocol will support just two operations, a span-size read and a span-size write.

For read operations, the request packet contains just the header. For write operations, the header will be followed by the span data to be written.

After receiving a request packet from the client, the memory blade should respond with a response packet which starts with a header as detailed in Table 6.2. The response header contains a response code to indicate whether it is a response with data (for read requests) or a response without data (for write requests). The header also includes a transaction ID, which should match the transaction ID of the request it is responding to. For read responses, the header will be followed by the data requested.

6.2 Hardware Design

The design of the memory blade controller is shown in Figure 6.1. The memory blade receives network packets from the output tap of the NIC, which is configured to redirect all remote memory requests based on the EtherType. When a packet comes in from the NIC, it first goes into the protocol handler, which parses the header into a single bundle and sends it, along with the data (if a write), to the router. The router then uses the header to route the request to one of several banks, each of which is responsible for a distinct set of spans (the spans are striped across the banks). At the inflow of the bank is an allocator, which chooses one of the available workers to allocate the request to. Each worker handles a single request at a time, and contains a span-sized SRAM buffer to store the data to be read/written. To access memory, each bank gets a dedicated port into the system bus. The various workers in each bank arbitrate for access to this port. Once the memory accesses have completed, the workers send responses back through the allocator and router to the protocol handler, which constructs a response header and sends it (along with the data in the case of read responses) out to the network through NIC input tap.

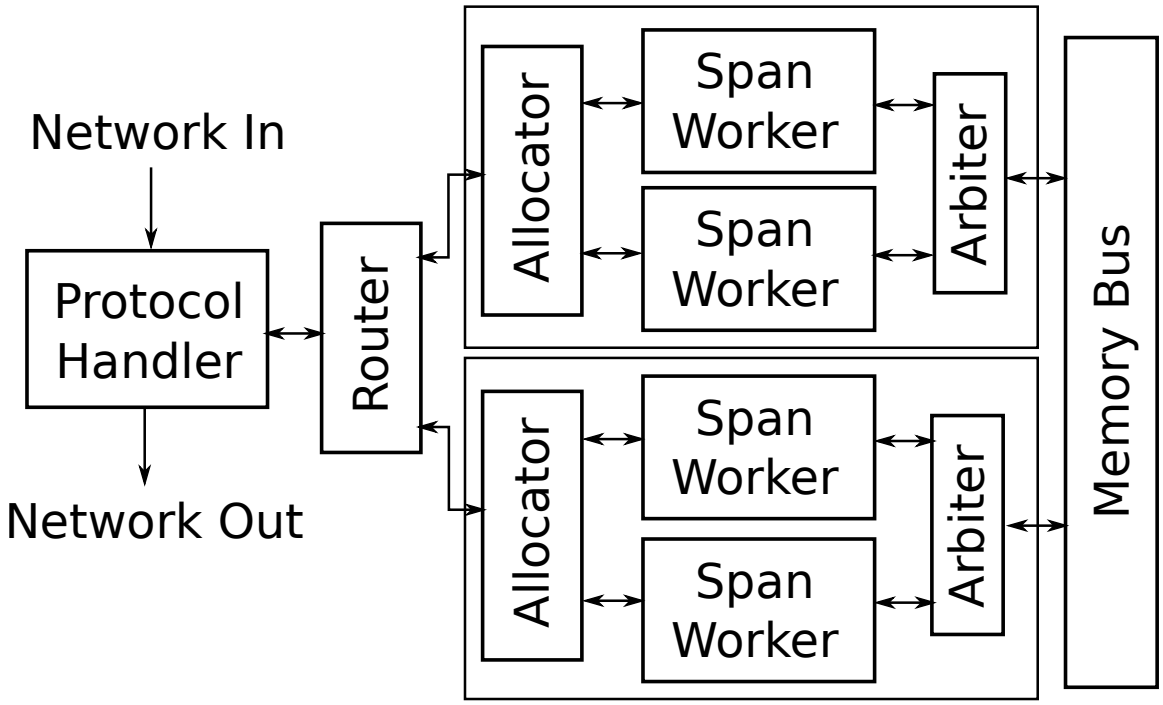


Figure 6.1: Memory Blade Design

The design of the remote memory client is shown in Figure 6.2. The CPU controls this device through MMIO by writing to registers for the opcode, the span ID on the memory

blade to read/write from, the MAC address of the memory blade, and either the source or destination address in local DRAM (for writes and reads, respectively). Once these values are set, the CPU will read from a request register, causing a request to be sent to the request allocator, which will select a transaction ID that is not yet in-flight and return that ID in response to the MMIO read. Then the request, along with the selected transaction ID, gets sent to the request handler, which constructs Ethernet and remote memory request headers and sends them out to the network through the NIC input tap. In the case of write requests, the request handler will also read the write data from local DRAM and send that out to the network as well. From the request handler, the request is forwarded to the response handler, which stores the request in a table indexed by the transaction ID. Responses come to the response handler from the NIC output tap, and the header is parsed to determine the transaction ID, which is then used to look up the matching request in the table. If the response is for a read, the response handler also writes the data to the local DRAM according to the destination address stored in the request table. Once the transaction is complete, the response handler notifies the request allocator so that it can make the transaction ID available for reuse. The response handler will also send a completion to the controller, so that it can send an interrupt to the CPU to notify it that the request is complete. The CPU's interrupt handler should then read from the response register to acknowledge that the response was received and free up space in the queue.

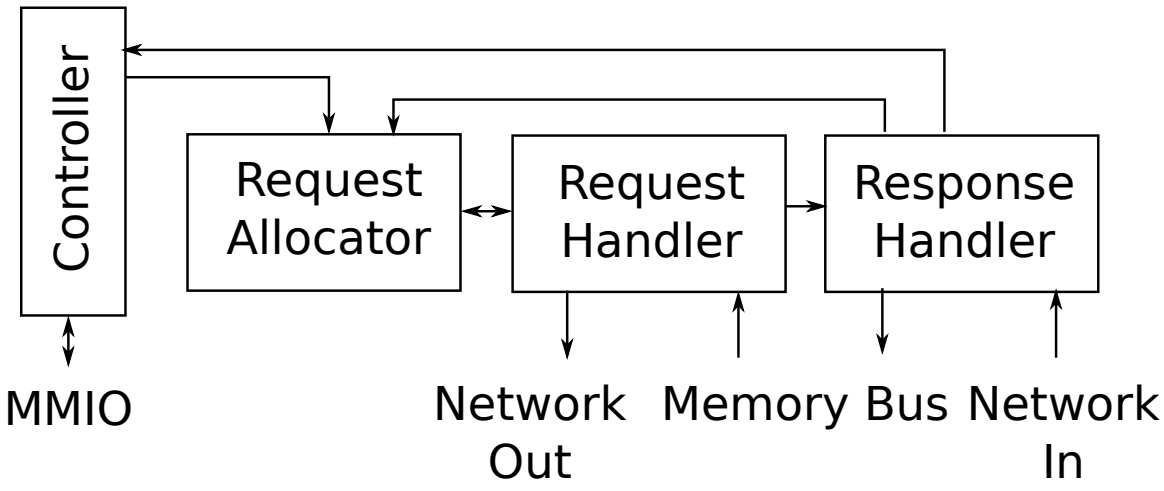


Figure 6.2: Remote Memory Client Design

6.3 Benchmarks

In this section, we will perform some micro-benchmarks on the remote memory client and memory blade system. We are primarily interested in the raw bandwidth and latency that can be achieved, as well as how this compares to a system in which remote memory accesses are triggered by page faults.

Baseline System

In order to have meaningful comparisons for our benchmarks, we must ensure that the parameters of the system being simulated match that of an existing system. The system we compare against are the Firebox-0 nodes, which are X86 servers connected via a Mellanox Infiniband fabric. The CPUs in these servers are based on the Intel Coffee Lake microarchitecture. The most important parameters to consider in our case are those related to the cache hierarchy and memory system. The WikiChip article on Coffee Lake [35] states that the microarchitecture features a 32 KiB L1, a 256 KiB L2, and 2 MiB/core L3. The Firebox-0 nodes are eight core systems, meaning their L3 should be 16 MiB. The access latency is stated to be 1.3 ns for L1, 4 ns for L2, 14 ns for L3, and 65 ns for main memory. We were able to confirm these figures for Firebox-0 by running the CCBench [5] tool on one of the machines. The CCBench tool performs pointer-chases on increasingly larger working sets to measure the latency of various points in the cache hierarchy. The results are shown in Figure 6.3. There are four plateaus in the average latency of a pointer access, corresponding to the L1, L2, L3, and main memory. The first plateau is at about 1.7 ns, the second at 4 ns, the third at 13 ns, and the last at 66 ns. The inflection points between these plateaus also correspond to the L1, L2, and L3 cache sizes of 32 KiB, 256 KiB, and 16 MiB, respectively.

Another parameter we would like to determine is the number of requests that can be sent in-flight from each cache. For that, we use the "band_req" benchmark from CCBench, the results of which are shown in Figure 6.4. This benchmark performs N parallel pointer chases and records the throughput. The point at which further parallelism fails to deliver increased throughput indicates the maximum number of in-flight requests the memory system can sustain. The results show that there is no meaningful improvement to throughput beyond 12 parallel threads of execution, so we can surmise that 12 is the maximum number of in-flight requests supported.

Bandwidth and Latency

The first performance figure we would like to measure is the raw bandwidth and latency between the remote memory client and the memory blade. For this microbenchmark, we run a bare-metal program that interacts directly with the remote memory client through MMIO and polls for completion. This way, we can be reasonably assured that the result will not be affected by CPU speed or kernel overhead. But to make sure the CPU overhead is as low as possible, we use a BOOM CPU instead of the default Rocket CPU. To measure

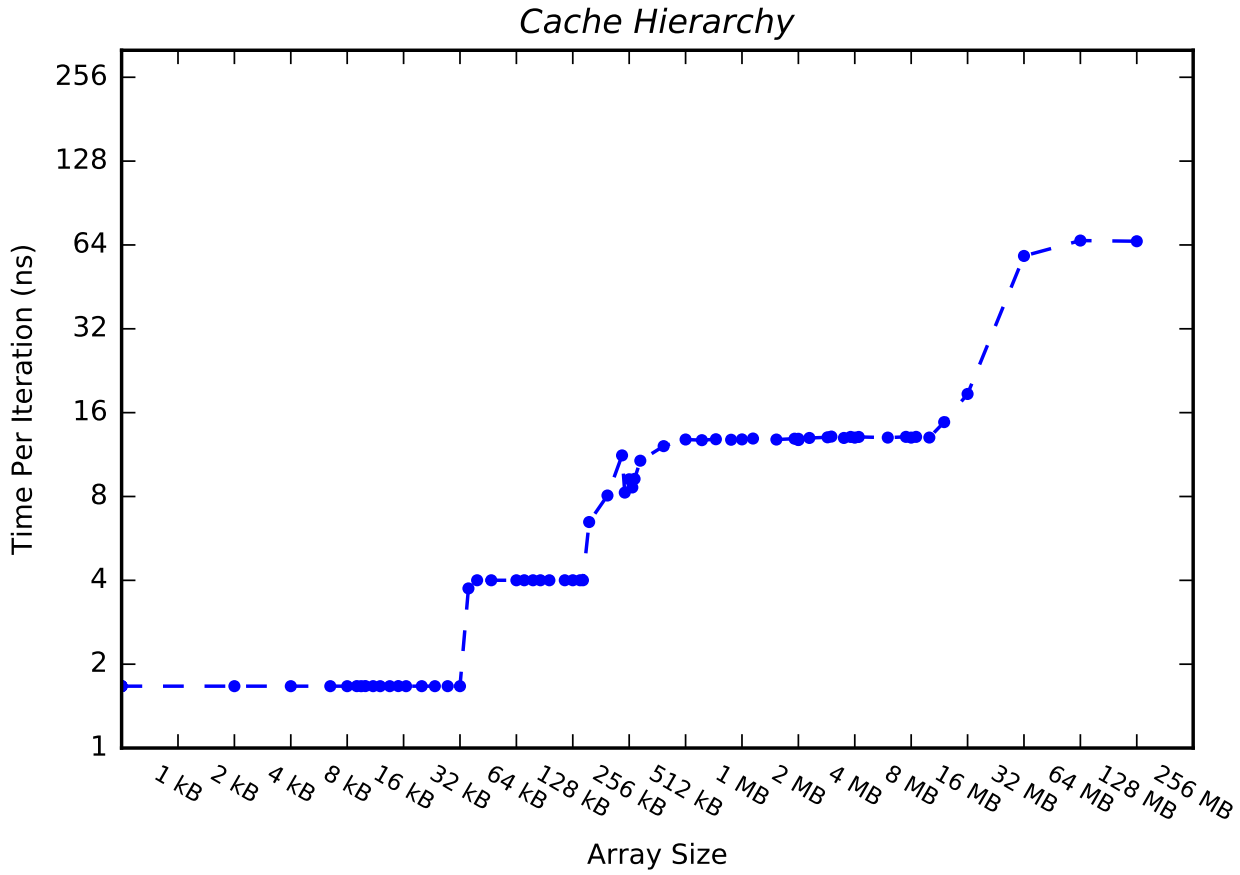


Figure 6.3: Firebox-0 CCBench results

bandwidth, we send as many write or read requests in-flight as possible and measure how long it takes to transfer a fixed number of spans between the compute node and the memory blade. To measure latency, we send single requests at a time. We configure the memory system to match the parameters of the Firebox-0 system by adding a 256 MiB L2 cache with four cache banks and three miss-status holding registers (MSHRs) per bank. The reason we use four banks is because the Coffee Lake memory system has 64 bytes per cycle bandwidth between L1 and L2, and our memory ports are 16 bytes wide. Since the remote memory client performs DMA into the L2 directly, bypassing the L1, we need not concern ourselves with the L1 parameters. We do not have an LLC model with a similar latency to the Coffee Lake L3. Instead, we configure a fixed-latency memory model to have a latency of 61 ns, the difference between the latency of an L2 access and a main memory access on Firebox-0. On the memory blade, we do not configure any caches. Instead, the memory blade controller communicates directly with a fixed latency memory model with a latency of 51 ns (the difference between main memory and L3 latency in Coffee Lake). Both client and memory blade are configured to have two DRAM channels, the same as Coffee Lake. For this test,

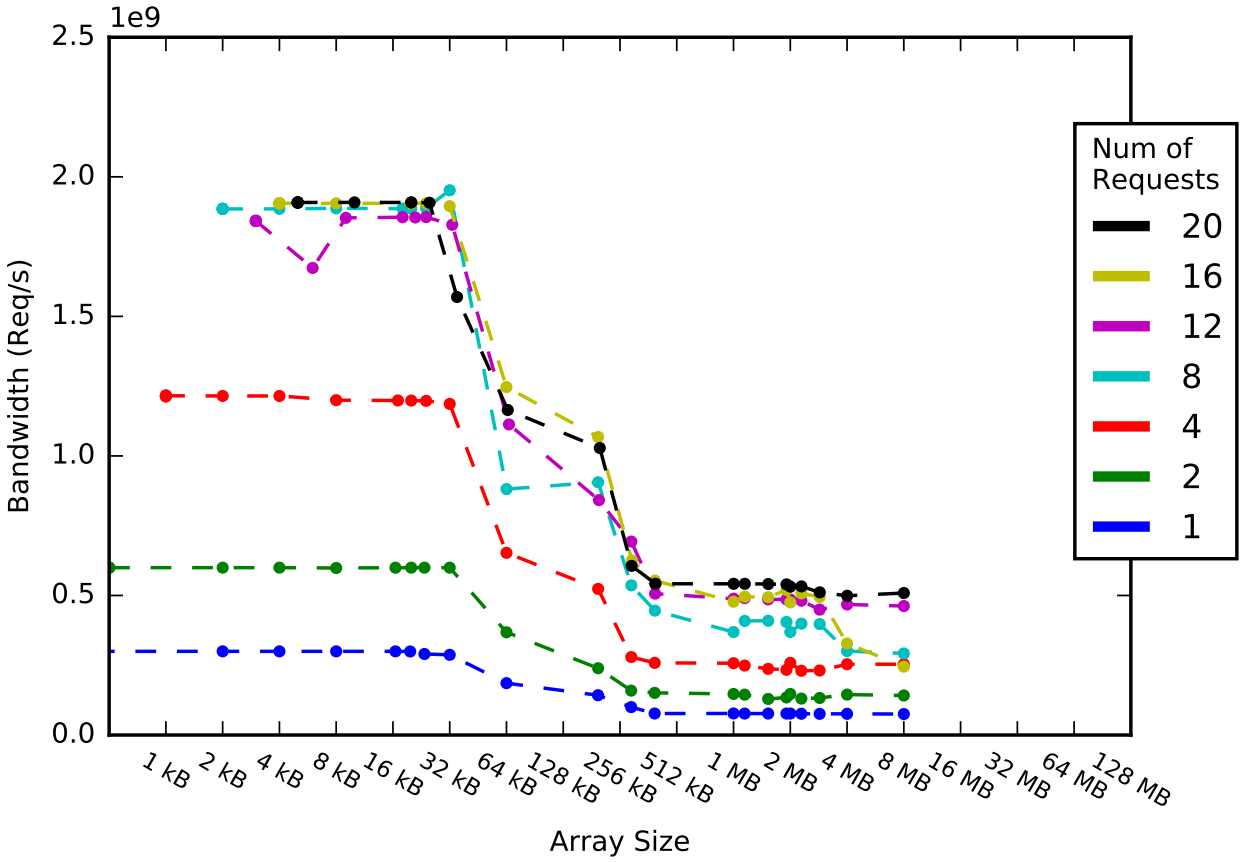


Figure 6.4: Firebox-0 band_req results

	Bandwidth	Latency
Write	90 Gbit/s	1224 ns
Read	92 Gbit/s	1224 ns

Table 6.3: Remote Memory Client Bandwidth and Latency Microbenchmark Results

we set the round-trip network latency to be one micro-second, the network bandwidth to 200 Gbits/s, and allow the remote memory client to send up to 32 requests in-flight. The results of the microbenchmark are shown in Table 6.3.

We expect the bandwidth of this microbenchmark to be constrained by the memory bandwidth, as there is a limited number of MSHRs. With 12 MSHRs, 64-byte cache lines, and a memory latency of 65 ns, the ideal bandwidth would be:

$$\frac{12lines \times 64bytes/line \times 8bits/byte}{65ns} = 94.5Gbits/s$$

And indeed we do see the measured bandwidth to be close to 94.5 Gbits/s. The measured bandwidth is a bit lower because there is some setup time between a response being received at the remote memory client and the next request being queued by the CPU.

The measured latencies are consistent with our various latency and bandwidth parameters. The bulk of the latency is the 1 μs it takes to traverse the network. In addition, it takes 64 ns to complete the first round trip to memory on the client and another 51 ns to do the same on the memory blade (there is no L2 or L3 cache on the memory blade, so that portion of the memory latency isn't considered). Finally, there is a serialization latency of $\frac{1024bytes \times 8bits/byte}{96Gbits/s} = 85ns$. This all adds up to 1200 ns. The remaining 24 ns can be attributed to the latency of CPU to remote memory client interactions.

Page Faults

Now that we know the raw bandwidth and latency of the remote memory client, we would like to see how much overhead would be introduced by mediating remote memory accesses through the virtual memory system. To this end, we will run a pointer-chasing benchmark (much like the one used in CCBench) on the client node's BOOM core. But instead of having the benchmark use main memory directly, we instead map remote memory into the virtual address space through a custom character device driver. On a page fault, the driver allocates a page in physical memory, copies data from the corresponding page on the memory blade to the local page, and then maps that page to the userspace program's address space. When the page is unmapped, the data will be copied back to the memory blade. In order to reduce overhead as much as possible, the driver eschews some typical features such as switching context while the remote memory requests are being made and using interrupt handlers to process responses. Instead, on a page fault, the driver sends the remote memory request and polls for the response. This has the drawback that the page fault cannot be interleaved with work from other threads, but we are not using multiprogramming in this benchmark in any case. We gather results for this benchmark while varying the round-trip network latency from 1 μs up to 5 μs. We expect the virtual memory overhead to be independent of the network latency, so a scatter plot of the results should show a linear trend with unit slope. The results of the pointer chase benchmark (Fig. 6.5) do indeed show such a trend, with a y-intercept of about 1.5 μs. This means that the page fault overhead more than doubles the overall latency of a remote memory access.

6.4 Chapter Summary

In this chapter, we show the design of the server portion of our system: the memory blade. We also build and evaluate a baseline client design which relies on explicit RDMA calls from the CPU. Through our evaluation, we found that RDMA adds very little overhead beyond the latency of the network and memory. We also found that RDMA can match the bandwidth of local memory if enough requests are sent in-flight. The issue, however, is that this low

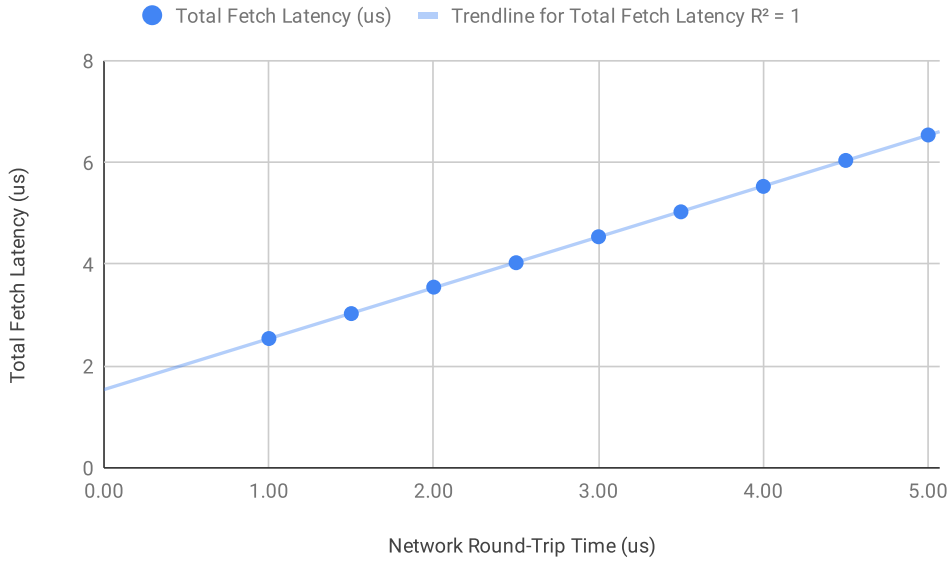


Figure 6.5: Latency of Remote Memory Access via Page Fault

overhead only applies if the application programmer manually issues the RDMA requests. Hiding this complexity through the virtual memory system produces an unacceptably high overhead. In the next chapter, we will show how hardware-managed DRAM caching can maintain the low overhead of RDMA but still make remote memory access transparent to the application.

Chapter 7

DRAM Cache Design

As we saw in the previous chapter, managing transfers between local and remote memory using software requires either directly exposing memory management to the application programmer or unacceptably high overheads from page faults. For a remote memory access solution that is both transparent to the application programmer and low overhead, we explore a fully hardware-managed solution. Like the virtual memory-based system from the previous chapter, a hardware-managed DRAM cache will make remote memory access transparent to the application. Unlike the virtual memory-based system, however, the hardware-managed DRAM cache will not have the overhead of interrupting the CPU to run a page fault handler. Instead, the only additional overhead will be a single access to DRAM to retrieve metadata. In this chapter, we will describe in detail the microarchitecture of our DRAM cache implementation and the automatic prefetcher that accompanies it. We will then present results of our initial evaluation, which includes a raw bandwidth/latency microbenchmark and an example SAXPY workload. The microbenchmark will also demonstrate the effects of some of the microarchitectural features and tradeoffs discussed in Chapter 3.

7.1 System Overview

The DRAM cache connects to the rest of the SoC as shown in Figure 7.1. As in the standard Rocket Chip SoC, the L1 instruction and data caches of each CPU connect to a shared multi-bank L2 cache. Unlike the standard SoC, the back side of the L2 cache connects not only to the regular DRAM channels, but also to the DRAM cache controller. The cache controller connects to its own dedicated DRAM channel, which it uses to store both cache data and metadata. The cache is refilled by requesting data from the memory blade through the Ethernet network. To access the network, the cache controller connects to the NIC through the tap-in/tap-out interfaces. Between the L2 and the cache controller there is also an automatic prefetcher, which detects accesses to sequential cache lines and sends prefetch requests for subsequent cache lines.

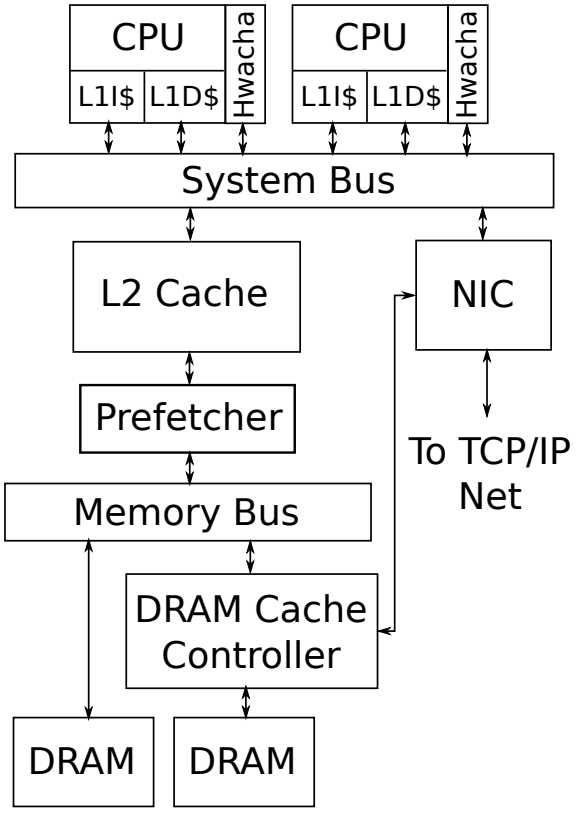


Figure 7.1: DRAM Cache within SoC

7.2 Cache Controller

The design of the DRAM cache controller is shown in Figure 7.2. The cache is organized into multiple cache channels, which are in-turn organized into multiple cache banks. Memory requests are routed to the banks based on their address, with cache lines striped across banks.

The Life of a Cache Request

When a memory request arrives at a bank, it is accepted by the scheduler, which stores the request metadata (opcode, address, size, and transaction ID) in the request buffer. If it is a put request, the scheduler will also store the data and write mask of each beat in the put buffer. The scheduler directs all of the events required to fulfill the request. It first sends a request to the metadata handler to determine whether or not the line is present in the cache. The metadata is kept in the DRAM itself, so the metadata handler sends a request to DRAM for the metadata associated with the set the cache line belongs to. Upon

receiving the response, the metadata handler checks the tags for each cache way. If there is a match, it returns the matching way to the scheduler. Otherwise, it randomly chooses a way to evict and returns the evicted way. If the access is a hit, the scheduler directs the local get handler or the put handler to read or write, respectively, the requested data from/to DRAM and send the response back to the requester. However, if the request is a miss, the scheduler sends a request to the remote access handler to obtain the data from remote memory. When the data comes back from remote memory, it goes to the put handler, which writes it to the local DRAM. If the request is a put, the put handler will also dequeue the put data from the put buffer and merge it with the remote data. If the request is a get, the remote data is also sent to the remote get handler so that the requested data can be sent in response. Additionally, if a way is being evicted and the cache data is dirty, the scheduler sends a request to the write-back unit to write the dirty data back to remote memory. The write-back unit must also send requests to the remote access handler to do this. Once the evicted cache line has been written to remote memory and the data for the new cache line is returned, the scheduler writes the new data to DRAM and then sends a response to the requester. Each scheduler can track multiple independent in-flight requests. The necessary state for tracking the progress of each request is held in an MSHR. Events like metadata and remote access responses are correlated to the MSHRs through transaction IDs.

Remote Access Handling

The main responsibility of the remote access handler is to format the data for the network packet header. The body of the request (in case of writes) is provided by the scheduler or write-back unit. First, the logical address that is passed to the cache in the request must be mapped to the network address of a memory blade (in this case, a MAC address) and the physical address on that blade. These mappings are stored in the extent table. The logical address space is mapped in 1GB sections we call extents. This amortizes the cost of allocation and simplifies how we store the mappings. As Rocket and Boom both use Sv39 virtual memory systems, which has 39-bit virtual and physical addresses, the logical address space can only go up to 512 GB. We further limit the address space of the DRAM cache to 128 GB so that there will only be 128 extent table entries, each of which is only 64 bits. This is small enough to fit comfortably in SRAM, so there is no need to store the table in DRAM and perform another expensive access to get the mapping. Once the remote access handler gets the mapping from the extent table, it sends the request header and (in the case of write-backs) forwards the write data. It also receives responses from the network and routes the data to the bank that sent the corresponding request. The wire protocol used by the DRAM cache is the same as the one used by the remote memory controller in Chapter 6.

DRAM access arbitration

Each cache channel has its own dedicated DRAM channel for storing data and metadata. The banks in the channel share access to DRAM through the channel arbiter, which is

organized as a ring network. Each channel has its own write-back handler, which handles write-back requests for all banks in that channel. All of the channels share a single remote access handler. Shared access to the write-back handlers and remote access handler are also arbitrated by ring networks. Each bank has its own metadata handler, which shares access to the channel arbiter port with the scheduler. The metadata handler and scheduler's access to the channel arbiter port are arbitrated by a crossbar.

Metadata Caching

To reduce the latency of metadata access for cache hits, the metadata handler can optionally contain a small SRAM cache that can cache the metadata for recently accessed cache lines. This metadata cache is organized as a direct-mapped, write-through cache. It is indexed by a subset of the set index bits and stores the full tag (including the cache tag and the remaining set index bits), present/dirty bits, and the way in the set at which the line is stored. This way, if the address matches the full tag and the present bit is set, the metadata handler has all the information it needs to send to the requesting tracker without accessing DRAM at all. If the cache line is not found in the metadata cache or the metadata is updated by the tracker, the new metadata can be written into the metadata cache without checking if it is already occupied, as the cached metadata is always clean with respect to the metadata stored in DRAM. The metadata cache is also useful for the cache replacement policy. If a cache line's metadata is present in the metadata cache, it must have been the most recently accessed cache line in its set. When choosing a way for eviction, the replacement policy will randomly choose a way other than the one stored in the metadata cache. This allows some degree of LRU behavior while avoiding writing to DRAM to update LRU state on cache hits.

7.3 Prefetcher

The final component of the DRAM caching system is the prefetcher. Because remote memory accesses have long latencies, we provision a large number of cache MSHRs to achieve decent bandwidth. However, the L2 cache has a much smaller number of MSHRs, so if we relied solely on CPU accesses to trigger DRAM cache refills, the DRAM cache MSHRs would not be well utilized. The solution is a prefetcher that automatically detects when the L2 is requesting a sequential stream of cache lines and sends prefetches for lines further ahead in the stream.

The design of this prefetcher is shown in Figure 7.3. Memory requests come in from the L2 and are forwarded to the DRAM cache. If the memory request is a cache line-sized read request, the prefetcher consults a fully associative predictor table to determine whether it is part of a sequential stream. Each predictor way contains a set of block addresses, a counter, and a timer. If the cache line address matches one of the block addresses, that block gets set to the next one in the sequence, and the way counter is incremented. If the counter has reached a set threshold, the cache line is determined to be part of a stream,

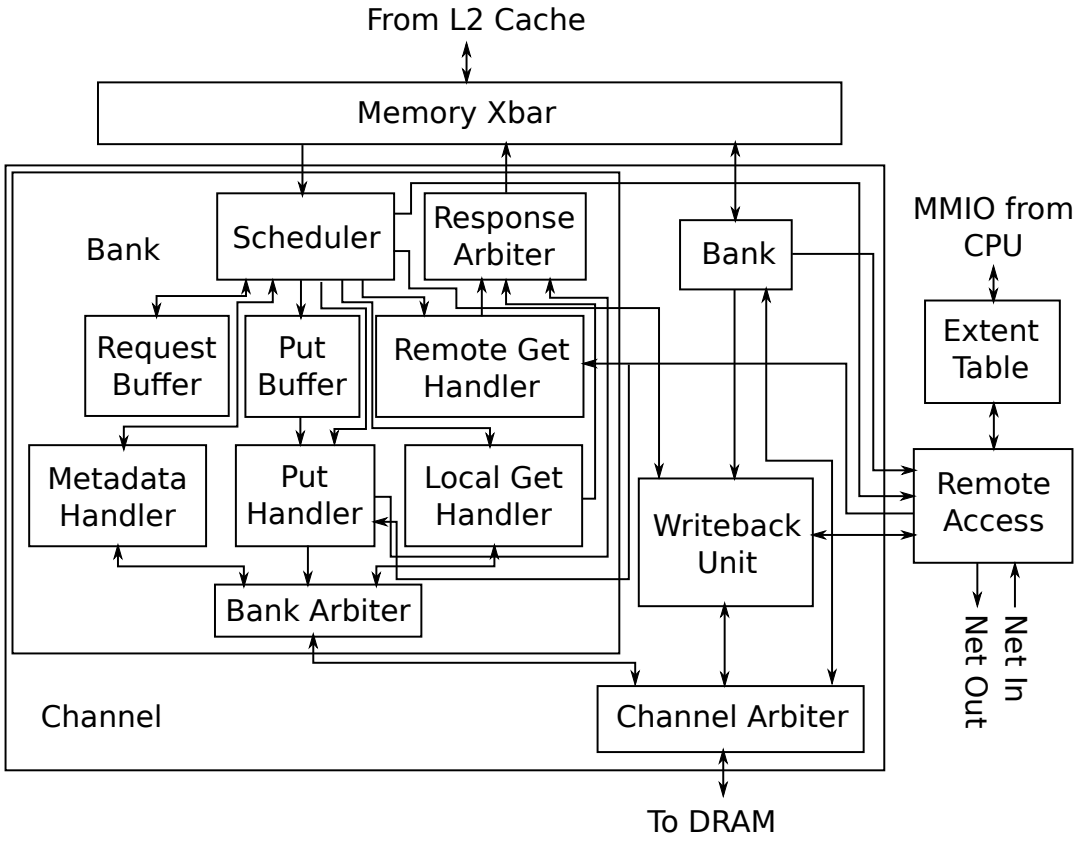


Figure 7.2: DRAM Cache Design

and a prefetch request is sent to the prefetch worker associated with the predictor way. If there is no match, one of the predictor ways is chosen through pseudo-LRU replacement and its blocks array is filled with the next few cache lines after the requested one. On each match, the timer is reset to a runtime-configured timeout. On every cycle when there is no matching request, the timer counts down. If it expires before a match comes in, the counter is zeroed and the predictor way will have to be retrained. Upon receiving the first request from the predictor, the prefetch workers send prefetch TileLink requests to the DRAM cache for the subsequent cache lines in the sequence up to a set limit. Once the limit is reached, the worker ceases sending prefetches until another match is detected by the predictor. This ensures that prefetches don't run too far ahead of the actually requested cache lines and create cache pollution.

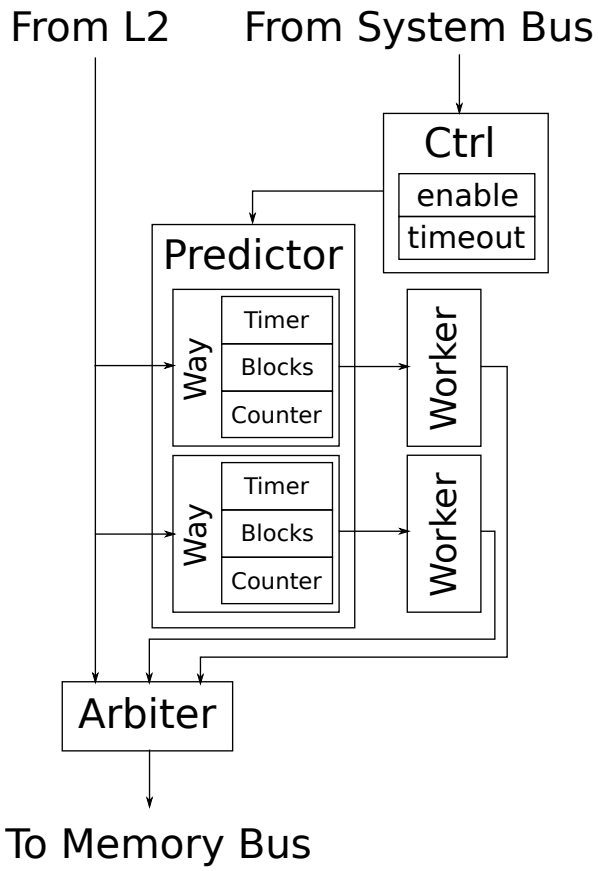


Figure 7.3: Prefetcher Design

7.4 Microbenchmark

As with the RDMA-based system, we first run some microbenchmarks to characterize the bandwidth and latency of the DRAM cache system. To collect this data, we use a specialized peripheral called MemBench, which schedules memory accesses and records cycle timing completely in hardware. This avoids depending on the memory scheduling of the CPU pipeline, which can introduce additional latency overheads or bandwidth constraints. For this microbenchmark, we wish to record the bandwidth and latency of the DRAM cache in two cases: the first with none of the data cached (requiring a remote memory access for every cache line) and the second with all of the data already cached in local DRAM. The memory system is mostly configured the same as it was for the remote memory client system in Chapter 6, but since a DRAM cache access requires two round-trips to memory (one for metadata and one for data), we double the number of L2 MSHRs to allow approximately the same local DRAM bandwidth to be achieved. Additionally, the 61 ns latency for DRAM is

redistributed, with a 10 ns latency pipe between the prefetcher and DRAM cache to represent the L3 latency and 51 ns between the cache and the DRAM channels. We build four DRAM cache configurations, which use either 64-byte or 1024-byte cache lines and either have or don't have metadata SRAM caches. The parameters for these four configurations are shown in Table 7.1. Every configuration allows up to 14336 bytes of data to be in flight at a time. This number was chosen because it should theoretically allow the same bandwidth to remote memory as the baseline configuration can achieve with local memory. The expected latency of a remote access is $61ns + 1000ns + 51ns = 1112ns$, and $\frac{14336bytes \times 8 \frac{bits}{byte}}{1112ns} \approx 103Gb/s$.

Parameter	Config 1	Config 2	Config 3	Config 4
Cache line size (bytes)	64	64	1024	1024
Cache channels	2	2	2	2
Banks per channel	16	16	1	1
Trackers per bank	7	7	7	7
Metadata Cache Rows (Per Bank)	256	0	16	0

Table 7.1: DRAM Cache Parameters

We run three sets of microbenchmarks for each DRAM cache configuration: one using an empty cache without prefetching, one using an empty cache with prefetching, and one using a cache with all of the data already present. For each set, we measure the bandwidth and latency of both reads and writes. The results are shown in Table 7.2. The "Local" configuration is just the remote memory client configuration from the previous chapter. In general, the read and write latencies in each group were similar to each other. For the local DRAM, we see that the latency for both reads and writes is 65.3 ns, which is consistent with the latency we have configured the memory system to and with the latency we observed on the X86 system. For DRAM cache hits, we see a latency of about 135 ns. The breakdown of this latency can be seen in Table 7.3. The main components of the cache hit latency are the "L3 delay" and two DRAM accesses, one for the metadata and one for the data. The cost of a DRAM access is a combination of the 51 ns DRAM latency and the cost of traversing the channel's ring network. For DRAM cache misses, we see a latency of 1153 ns for 64-byte lines and 1233 for 1024-byte lines. The breakdown for the latency of cache misses can be seen in Table 7.4. The additional latency in the 1024-byte configuration can be explained by added serialization latency. The main components of cache miss latency are the "L3 delay", a local DRAM access for retrieving metadata, a 1 μs network traversal, and a DRAM access on the memory blade. As expected, the latency of uncached accesses with prefetching is in between the two, but much closer to that of a cache hit.

Knowing the latency of the various types of accesses, we can make sense of the read bandwidth of the 64-byte line configurations. In the case of local accesses, cache hits, and cache misses without prefetching, it is simply the number of L2 MSHRs multiplied by the size (in bits) of a cache line divided by the latency. In the case of cache misses with prefetching,

it is the number of DRAM cache MSHRs multiplied by the cache line size divided by the (non-prefetched) miss latency. In the benchmark without prefetching, the configuration without metadata caching has similar bandwidth to the configuration with metadata caching. However, in the benchmark with prefetching, the configuration with metadata caching had higher bandwidth. This is because the metadata cache can keep the metadata for recently prefetched lines in SRAM, saving the read request an additional round-trip to DRAM.

For the 1024-byte cache line configurations, the bandwidth for uncached reads without prefetching is slightly higher, but the bandwidth for cached reads and uncached reads with prefetching is much worse. This is mainly because the cache line size in the L2 is kept at 64 bytes. Thus, each L2 miss only requests a portion of the DRAM cache line. This is beneficial for uncached reads without prefetching, because the first L2 miss for a DRAM cache line will trigger a miss and the rest will be hits. However, for cached or prefetched reads, this is detrimental because it increases the number of set conflicts. When an MSHR is occupied with a certain cache set, it blocks other MSHRs from accepting requests for the same set until its current transaction completes. Sequential L2 lines will all be in different DRAM cache sets if the DRAM cache lines are 64 bytes, but in the same set if the lines are 1024 bytes. Sometimes, requests to the same DRAM cache line can be merged into a single MSHR, but there are still times when there is no choice but to back-pressure the TileLink request from the L2. Because of limited buffer space in the network, this can end up creating head of line blocking, which stops a request for a non-conflicting cache line from being routed. This leads to periods when not all MSHRs are active, even when there are still requests waiting to be processed.

The write bandwidth for the 64-byte cache line configurations is always around 161.6 Gb/s, no matter where the data is. This is because we are using full cache line writes, so the L2 can simply overwrite the entire line in the data array without sending a refill request to local DRAM or the DRAM cache. The only requests the DRAM cache sees are evictions from the L2. Since these are also the size of a full cache line, the DRAM cache can similarly overwrite the line in DRAM without requesting the existing data from remote memory. This is not the case for the 1024-byte cache line configurations, which will need to access remote memory for L2 evictions that result in DRAM cache misses. This makes the uncached write bandwidth similar to the uncached read bandwidth. Prefetching does not help in this case because the auto-prefetcher only triggers on L2 refills and not L2 evictions.

Because the DRAM cache configuration with 64-byte cache lines and metadata caching has the best performance by far, the remainder of our evaluation will focus solely on this configuration.

7.5 SAXPY Benchmark

Now that we know the raw performance of the DRAM cache, we would like to determine how well it supports interleaving data access and computation. To do this, we use a SAXPY benchmark. This is an embarrassingly parallel workload with a very predictable memory

Configuration	Benchmark	Bandwidth (Gb/s)	Latency (ns)
Local	DRAM Reads	92.7	65.3
	DRAM Writes	100.2	65.3
64-byte blocks / Metadata Cache	Uncached No Prefetch Reads	10.6	1156
	Uncached No Prefetch Writes	161.0	1156
	Uncached Prefetch Reads	87.1	144.0
	Uncached Prefetch Writes	160.8	142.4
	Cached Reads	90.4	134.7
	Cached Writes	161.6	134.7
64-byte blocks / No Metadata Cache	Uncached No Prefetch Reads	10.6	1156
	Uncached No Prefetch Writes	162.4	1156
	Uncached Prefetch Reads	80.0	144.3
	Uncached Prefetch Writes	161.1	143.6
	Cached Reads	90.6	134.3
	Cached Writes	164.1	134.3
1024-byte blocks / Metadata Cache	Uncached No Prefetch Reads	12.5	1233
	Uncached No Prefetch Writes	18.1	1233
	Uncached Prefetch Reads	40.1	132.0
	Uncached Prefetch Writes	18.8	133.6
	Cached Reads	64.2	132.6
	Cached Writes	24.9	132.7
1024-byte blocks / No Metadata Cache	Uncached No Prefetch Reads	12.5	1233
	Uncached No Prefetch Writes	18.1	1233
	Uncached Prefetch Reads	46.7	132.0
	Uncached Prefetch Writes	23.3	133.6
	Cached Reads	72.1	132.6
	Cached Writes	65.8	132.7

Table 7.2: MemBench Results

access pattern, so it should be able to drive the memory system at close to its full bandwidth. We implement two versions of this benchmark, one for X86 using AVX2 SIMD instructions, and the other for RISC-V using Hwacha vector instructions. Though the programming models of these two extensions are slightly different, they are both data-parallel floating point ISAs, which makes them comparable for this benchmark. The X86/AVX2 version runs on a FireBox-0 node and the RISC-V/Hwacha version runs on a FireSim simulation. To match the throughput of the 256-bit AVX2 instructions, we configure Hwacha to have two vector lanes, each of which is 128 bits wide. We show three different configurations of the RISC-V benchmark here, one running on local DRAM, one running on the DRAM cache without prefetching, and one running on the DRAM cache with prefetching. For the

L2 Delay	3 ns
L3 Delay	10 ns
Metadata memory NoC traversal	6 ns
Metadata DRAM access	51 ns
Data memory NoC traversal	6 ns
Data DRAM access	51 ns
Miscellaneous buffer delays	8 ns

Table 7.3: Cache Hit Latency Breakdown

L2 Delay	3 ns
L3 Delay	10 ns
Metadata Mem Access NoC traversal	6 ns
Metadata Mem Access DRAM latency	51 ns
Metadata Buffer Delay	3 ns
Remote Access NoC traversal	6 ns
Extent Translation	1 ns
Header Formatting	1 ns
Network Traversal	1000 ns
MemBlade DRAM access	51 ns
Misc. MemBlade delays	7 ns
NIC buffer serialization delay	3/43 ns
TileLink response serialization delay	3/43 ns
Misc. DRAM cache delays	8 ns

Table 7.4: Cache Miss Latency Breakdown

configurations using the DRAM cache, we also compare the performance with all the data already cached to performance with none of the data cached. All of these run on RISC-V Linux with the virtual memory pages locked before beginning the benchmark to exclude any page fault effects. However, unlike the remote memory client in the previous chapter, a DRAM cache page being mapped in virtual memory does not mean the data itself has been moved to local DRAM storage. The results of this benchmark are shown in Figure 7.4. We show the performance of the RISC-V benchmark normalized to the performance of the X86 baseline.

The RISC-V benchmark running on local memory achieves about 82% of the performance of the X86 baseline. The performance of the benchmark using the DRAM cache with all the data already cached has similar performance. But if the data is not already cached, the performance predictably drops quite drastically. However, if the prefetcher is enabled, the performance is almost the same as if all the data is cached, at 77% of the X86 baseline. One interesting result is that using the prefetcher when the data is already in the cache causes the

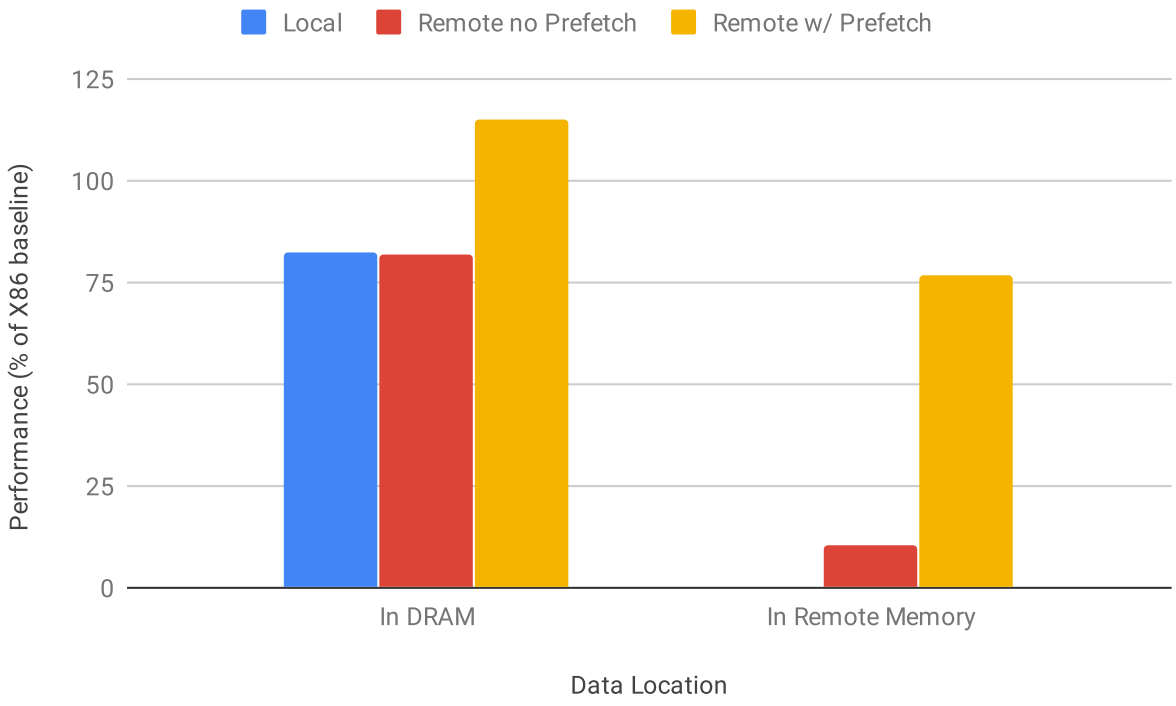


Figure 7.4: SAXPY Results

performance to be much higher than that of local memory. This is because of the interaction between the prefetcher and the metadata cache. Even though the access is a cache hit, the prefetch still pulls the metadata into the metadata cache, allowing the actual read access to skip the first DRAM access.

7.6 Chapter Summary

In this chapter, we showed the design of the hardware-managed DRAM cache and how it adds very little overhead to remote memory accesses. According to our microbenchmark, with a sufficient number of MSHRs and good prefetching, the DRAM cache can achieve a read bandwidth equivalent to that of local memory. This gives it excellent performance in workloads exhibiting a large degree of spatial locality, such as SAXPY. In the next chapter, we will show how the DRAM cache performs on more complex benchmarks.

Chapter 8

Evaluation of DRAM Cache System

In the previous chapter, we conducted a preliminary evaluation of the DRAM cache system using a SAXPY benchmark. This workload responds very favorably to automatic prefetching because its memory accesses are completely sequential and its control flow isn't data-dependent. In this chapter, we will further evaluate the DRAM cache system using more complex workloads with more irregular memory accesses and data-dependent control flow.

Datacenter workloads are usually separated into two categories: interactive and batch. Interactive workloads are applications that respond to user requests, such as web page loads or database lookups. They perform a relatively small amount of work for each request and thus are quite latency-sensitive. Designers of interactive applications generally try to reduce both the average latency and the tail latencies. By contrast, batch workloads are initiated by the data center operator and perform a large amount of work without any user input. Some examples of common batch applications in data centers are search/database indexing and machine learning. These workloads are throughput-oriented and aren't so concerned with tail latency.

In this chapter we will use two benchmarks to drive our analysis: one representing interactive workloads and one representing batch workloads. The interactive workload is Memcached [9], a key-value store. The batch workload is a graph algorithm benchmark we developed ourselves called friends-of-friends, a re-implementation of Facebook's "People You May Know" feature.

8.1 Memcached

In-memory key-value stores are a common application in data centers. Web applications often use these services to cache the results of database queries in order to more quickly serve user requests. However, if the amount of data that must be cached exceeds the capacity of a single node, application designers must decide how to partition the cache among several servers. One way is to have each client statically partition keys between servers (most commonly by using hashes of the key). This has the benefit of being relatively fast, since

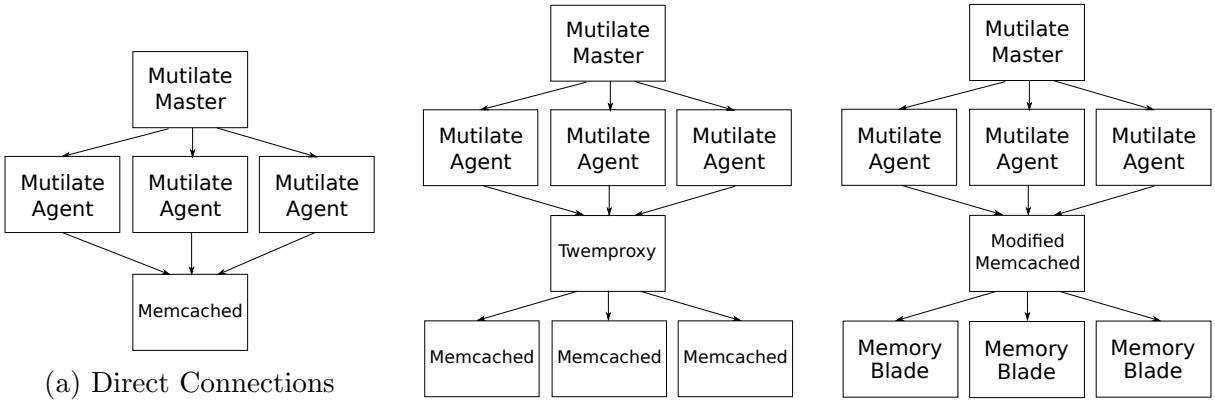
the clients are still connecting directly to servers. However, since each client must handle partitioning locally and all the clients' partitioning schemes must be the same, the partitioning scheme must be decided *a priori*, which makes it impossible to rebalance partitions if one is getting a larger share of traffic or data. An alternative is to use a proxy to forward requests to the key-value stores. Instead of connecting to the individual servers, the clients instead all connect to the proxy, which responds as if it was a single instance of the key-value store. This removes the need for all clients to agree on a partitioning scheme and allows rebalancing, but has the major downside of being much slower, since requests must make additional network hops and bottleneck at the proxy.

With our DRAM cache and memory blade designs, we can produce a distributed key-value store design that preserves the flexibility of mediating partitioning through a proxy but improves the performance. We modify the key-value store so that it allocates space in remote memory instead of local memory. We then map the logical address space to several different memory blades. While there will still be two network hops to satisfy a request in this design, the second hop will have lower latency, since it will not have the overhead of establishing a TCP connection and does not need to interrupt the memory blade's CPU to satisfy a request. It will still have the issue of being a central bottleneck, however.

Methodology

To explore these three designs, we use the Memcached key-value store [9], the Twemproxy proxy server [27], and the Mutilate benchmarking tool [24]. The Mutilate tool can send requests to a Memcached server (or set of servers) at a target queries per second (QPS). It can either do this directly or coordinate several agent processes to send the requests simultaneously. For our benchmarks, we will have a single coordinating process (the master) controlling a set of three worker processes (the agents). In the first set of benchmarks, all of the agents will connect directly to a single Memcached server (Fig. 8.1a). In the second set of benchmarks, we have an instance of Twemproxy sitting between the agents and the Memcached servers (Fig. 8.1b). Twemproxy will take a Memcached request, determine which server a key should map to, forward the request to the selected server, and forward responses back to the agents. From the agents' perspective, Twemproxy appears to be a single Memcached server. In the third set of benchmarks, we replace Twemproxy with a version of the Memcached server modified to use remote memory (Fig. 8.1c). This is a fairly simple modification, since Memcached already has the ability to allocate space for data from a preset pool of virtual memory. Ordinarily, this would come from a memory-mapped file. In our modified version, this pool is instead mapped to three different memory blades. Since we expect the address space to be allocated sequentially, we can implement some rough load-balancing by mapping consecutive pages to alternating memory blades (Fig. 8.2). In the final set of benchmarks, we use the same design as in the third set, but turn on automatic prefetching to the DRAM cache.

For this benchmark, we used SoC configurations with a BOOM out-of-order core on the compute nodes running the mutilate clients and the Memcached servers. The BOOM core



(a) Direct Connections (b) Twemproxy (c) DRAM Cache

Figure 8.1: Memcached Benchmark Configurations

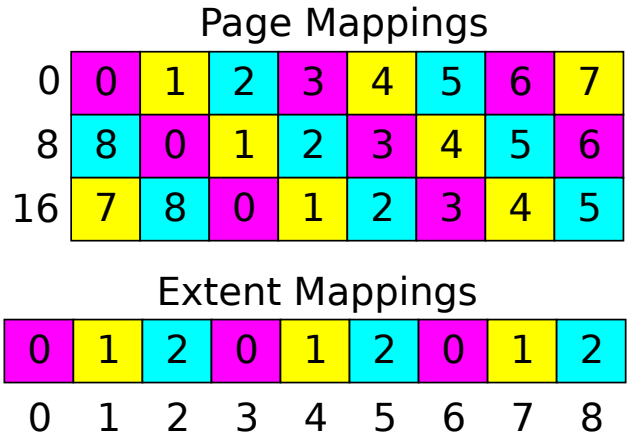


Figure 8.2: Mapping of extents to memory blades (bottom) and pages to extents (top)

has a non-blocking L1 data cache with 12 MSHRs.

For the DRAM cache version, we configure the DRAM cache to have slightly less space (56 MB) than what the Memcached instance uses (64 MB) so that there would be some evictions. Otherwise, the DRAM cache configuration is the same as the one used in the SAXPY benchmarks in the previous chapter.

Results

For the mutilate benchmark, we use the intended QPS as the independent variable and record four dependent variables: the actual QPS, the average latency, the 95% latency, and the 99% latency. Figure 8.3 shows a scatter plot of the intended QPS vs. the actual QPS for

all four sets of benchmarks. We expect the actual QPS to follow the intended QPS at first, but then flatten out once it reaches a limit. This is exactly what happens in the Twemproxy benchmark, with the QPS leveling off at around 24000 QPS. The direct benchmark and DRAM cache benchmarks level off more gradually, with the actual QPS falling behind the intended QPS but still increasing. The DRAM cache QPS levels off completely at around 65000, while the direct benchmark doesn't level off until around 112000 QPS. The relative order of maximum QPS among the four benchmarks roughly matches our expectation, with the DRAM cache faster than Twemproxy but slower than local memory Memcached. What was unexpected, however, was that prefetching into the DRAM cache made no improvement to the maximum QPS.

The average, 95th percentile, and 99th percentile latencies of the four benchmarks are shown in Figure 8.6. As expected, the local DRAM benchmark has the lowest average latency and the Twemproxy benchmark has the highest, with the DRAM cache average latency being in the middle. Also as expected, the DRAM cache latency was about twice that of the local DRAM latency. Somewhat unexpected is that, even though the DRAM cache has lower average latency than Twemproxy, it has higher tail latency. The maximum 95th percentile and 99th percentile latencies are higher for the DRAM cache than for Twemproxy. However, since these higher latencies occur at QPS higher than Twemproxy's maximum QPS, it's most likely the effect of additional queuing delay.

To gain more insight into why the DRAM cache behaves the way it does, we add synthesized print statements to the design and re-run a portion of the (non-prefetched) benchmark. Specifically, we run Mutilate with target QPS of 30000, 70000, and 120000. These print statements indicate when an MSHR was reserved by a request, when it was released, and whether a request was a cache hit or cache miss. Looking at the hit/miss information (Fig. 8.4) made it clear why prefetching does not improve performance. After the initial loading of the benchmark data at the beginning, the hit rate is generally quite high: between 80% and 100%. Therefore, the prefetches won't be able to improve performance. By counting the MSHR reservation and release statements, we can track the MSHR occupancy over time (Fig. 8.5), which reveals that the typical occupancy is around 4 MSHRs. This is much lower than the maximum occupancy of 24 (the number of L2 MSHRs), which explains why the maximum QPS for the DRAM cache benchmarks was approximately half that of the local DRAM benchmark. The latency of a DRAM cache hit is about twice that of a local DRAM access, and Memcached can't take advantage of the additional L2 MSHRs to send more simultaneous requests.

8.2 Graph Algorithms

We use a graph algorithm for our batch workload because graphs are a common data structure in modern web applications, especially in social networks. In addition, graph algorithms have some unique properties that make them interesting to evaluate. Unlike the SAXPY workload in the previous chapter, graph algorithms have irregular memory accesses and

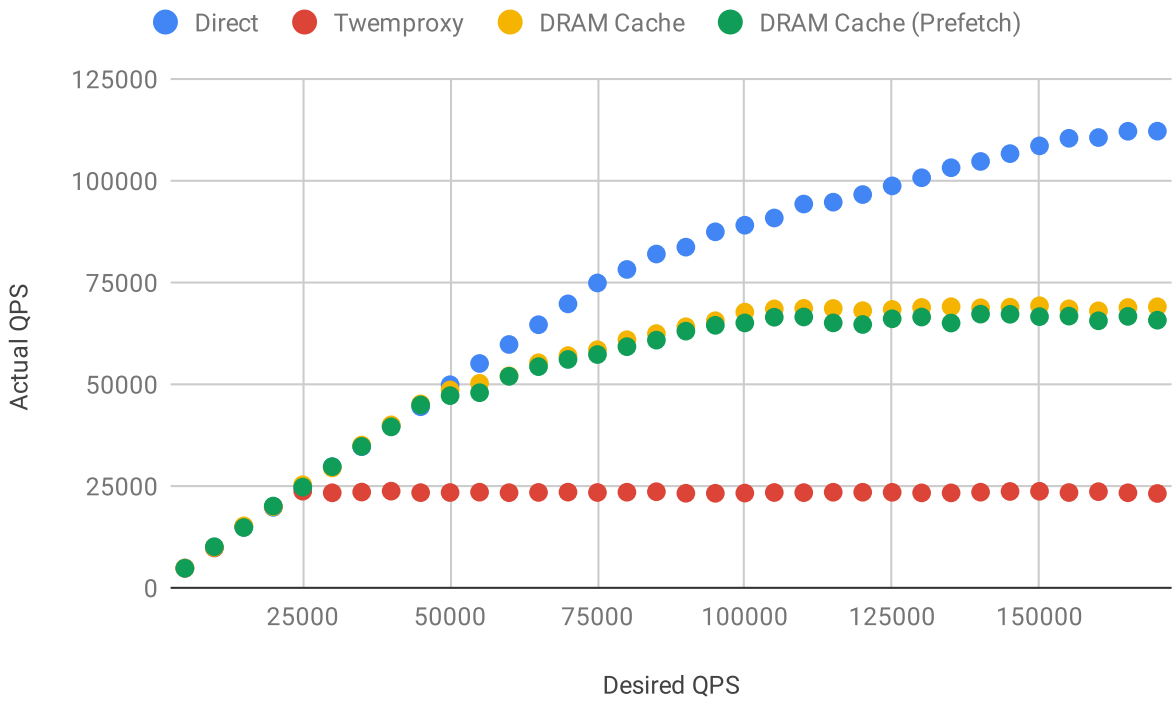


Figure 8.3: Mutilate Intended QPS vs. Actual QPS

data-dependent control flow. However, graph algorithms can still have a fair bit of spatial locality if the graph has a high average degree (i.e. each vertex has many edges). The specific graph workload we use is based on Facebook's "People You May Know" feature, which shows each user a list of other users who they are not friends with but who share mutual friends with them (henceforth, friends-of-friends or FOF). In our workload, we take in an undirected graph with vertices representing users and edges representing friendships. For each user, we follow the edges from that user's vertex to find all their friends. We then follow the edges from each of the friend vertices to find all the friends of those users that are not the original user nor one of the original user's other friends. During this traversal, we also count how many mutual friends the original user has with each friend-of-friend. In the final step, we sort the list of FOFs by descending number of mutual friends, using the user ID as a tie-breaker. The details of the algorithm can be seen in the Python reference implementation in Figure 8.7 and the graph diagram in Figure 8.8.

Methodology

To run the workload on the DRAM cache model, we implement the friends-of-friends algorithm in C, using a bit set for the "friends" set and a hash table with single-entry buckets

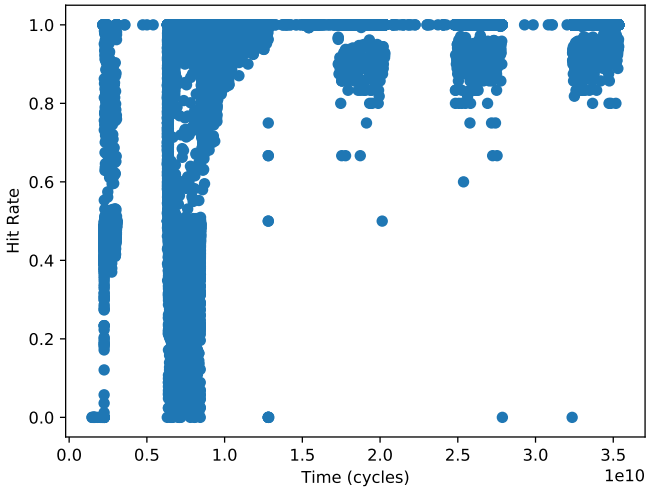


Figure 8.4: Hit Rate Over Time in Memcached Benchmark

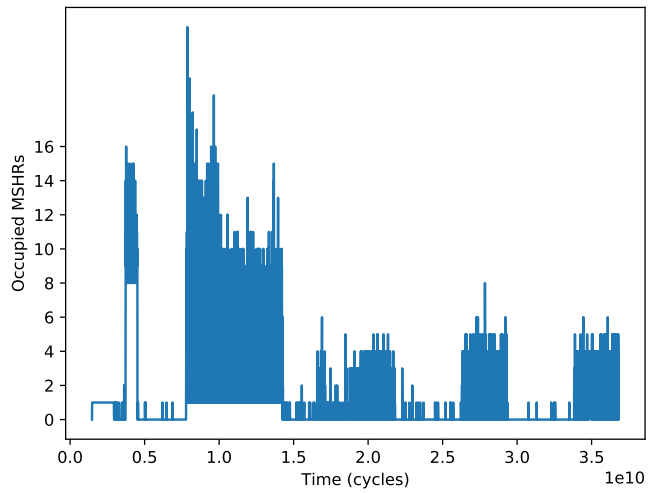
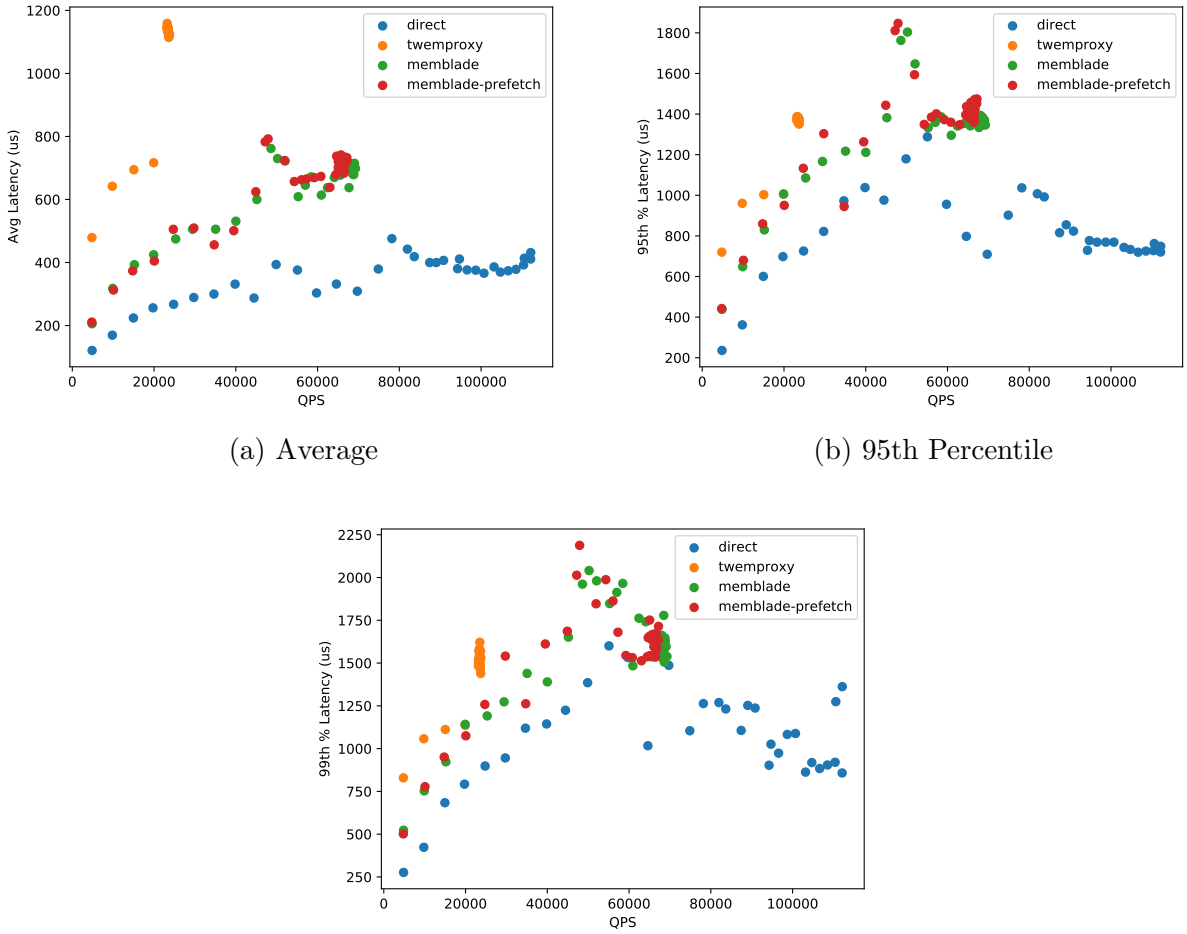


Figure 8.5: MSHR Occupancy Over Time in Memcached Benchmark

for the "mutuals" dictionary. To produce the "fofs" list, we allocate an array with a size based on how many FOFs were found when counting mutual friends. We then perform a second pass, copying any vertices with non-zero numbers of mutual friends to the "fofs" array. Finally, we use quick sort to sort the "fofs" array in-place.

For this benchmark, we use the anonymized Facebook friend graph produced for [28].



(a) Average

(b) 95th Percentile

(c) 99th Percentile

Figure 8.6: Mutilate QPS vs. Latency

We use the same SoC configurations used for the Memcached benchmark. We run three sets of benchmarks: one using local memory, one using remote memory without prefetching, and one using remote memory with prefetching.

Results

The results of this benchmark can be seen in Figure 8.9. As expected, using remote memory without prefetching significantly reduces performance, increasing the total runtime by 160%. However, using automatic prefetching brings the performance of remote memory closer to that of local memory, with a total runtime 60% slower than local memory.

```
FriendOfFriend = namedtuple("FriendOfFriend", ["id", "mutuals"])
def friends_of_friends(graph, user_id):
    friends = set([user_id] + graph[user_id])
    mutuals = defaultdict(int)

    for friend_id in graph[user_id]:
        for fof_id in graph[friend_id]:
            if fof_id not in friends:
                mutuals[fof_id] += 1

    fofs = [FriendOfFriend(fof_id, mutuals[fof_id])
                for fof_id in mutuals]
    fofs.sort(key = lambda x: (x.mutuals, -x.id), reverse=True)

    return fofs
```

Figure 8.7: Python reference implementation of friends-of-friends algorithm. The "graph" argument is a dictionary mapping a vertex ID to a list of neighboring vertices. The "user_id" argument is the ID of the user to find the friends-of-friends of.

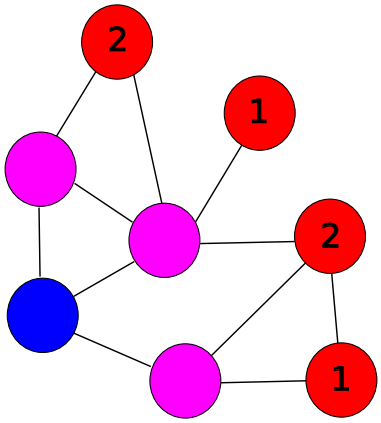


Figure 8.8: Sample friend graph showing the original user (blue), friends (magenta), and friends-of-friends (red). Numbers on FOFs indicate the number of mutual friends to original user.

As in the Memcached benchmark, we use synthesized print statements to determine the hit rate and MSHR occupancy when running the benchmark with remote memory and no

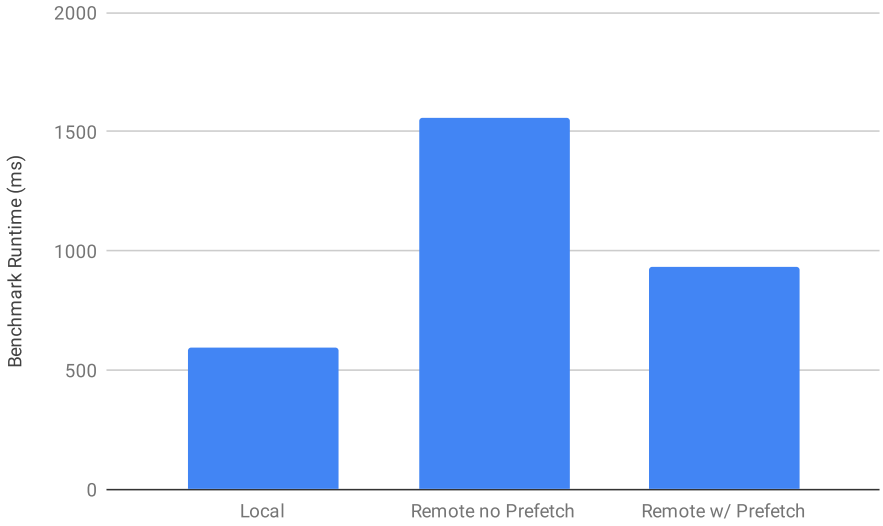


Figure 8.9: Friends-of-Friends Benchmark Results

prefetching. We can see in Figure 8.10 that the hit rate for the FOF benchmark is more varied than Memcached, often going as low as 20%. This explains why prefetching was effective for this benchmark but not for Memcached. The MSHR occupancy graph in Figure 8.11 shows that the FOF benchmark uses around 18 MSHRs at a time, demonstrating that it can take advantage of the additional L2 MSHRs.

Scaling

We saw in the previous section that the DRAM cache with prefetching performs a bit worse than local memory. But the DRAM cache provides one major benefit in exchange for this performance hit: better scaling. A system using the DRAM cache can easily scale to problem sizes larger than a single server's memory can hold simply by mapping the DRAM cache extents to multiple memory blades. We demonstrate this scaling by running the Friends-of-Friends benchmark with increasing graph sizes, expanding the graph by simply duplicating the base graph. We expect the amount of work to increase linearly with the amount of data. For this benchmark, each 1 GB extent will be mapped to a separate memory blade.

The results of this benchmark are shown in Table 8.1. As expected, when the size of the graph is doubled, the runtime doubles as well. The amount of memory used is still under 1 GB, so only one memory blade is involved. When the graph is quadrupled, the amount of memory used exceeds 1 GB, so two memory blades are used, but the runtime still scales linearly. This linear scaling continues even as the graph is made eight times larger, requiring the use of three memory blades.

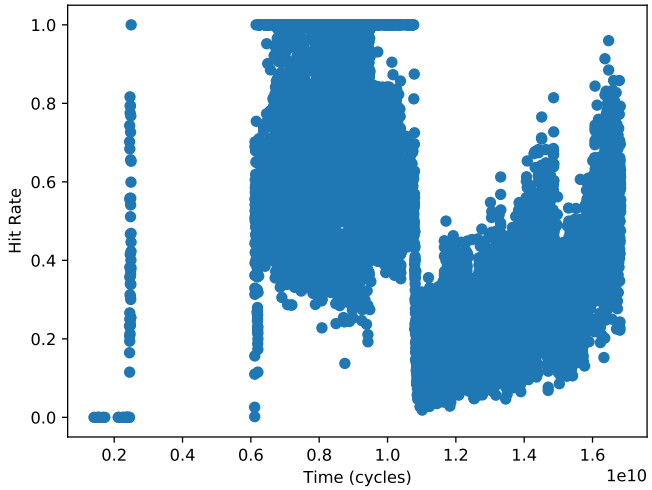


Figure 8.10: Hit Rate Over Time in Friends-of-Friends Benchmark

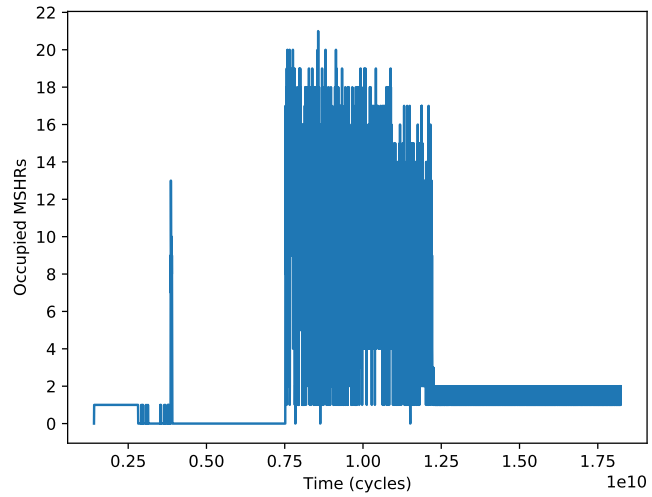


Figure 8.11: MSHR Occupancy Over Time in Friends-of-Friends Benchmark

8.3 Chapter Summary

In this chapter, we showed the results of running more complex benchmarks using the DRAM cache. The two benchmarks we used were Memcached, an example of an interactive workload, and Friends-of-Friends, an example of a batch workload. In the case of Memcached, we

Vertices	Memory Used (GB)	Runtime (s)	Runtime per Vertex (μs)
4039	0.32	0.94	234
8078	0.65	1.73	214
16156	1.33	3.29	203
32312	2.72	7.04	218

Table 8.1: Friends-of-Friends Benchmark Size Scaling

showed that the DRAM cache doubles the average latency and thus halves the maximum bandwidth of the system. While this may seem to suggest that the DRAM cache is not suited for interactive workloads, we note that the DRAM cache system's performance is still superior to that of a system using software-based proxies. This shows that the DRAM cache could be effective in cases where an interactive application is limited by a single node's memory and needs to be scaled up to multiple nodes. In the case of Friends-of-Friends, we showed that prefetching can greatly improve DRAM cache performance on batch workloads, even ones with more irregular memory access patterns. We also demonstrated that batch workloads running on the DRAM cache can easily scale as the size of the workload increases. The results of these benchmarks are some of the major findings and contributions of this work, which we will summarize in the next chapter.

Chapter 9

Conclusion

With technology improvements for conventional DRAM no longer keeping up with increasing demand for main memory in data centers, data center designers looking to decrease capital and operating costs must find ways of using existing DRAM more efficiently. One common way of improving data center resource utilization is disaggregation, in which a resource is organized into a global pool that can be accessed by multiple data center nodes over the network. While this technique is used extensively for secondary storage, it is not yet widely used for main memory. In this work, we explored in detail one approach to applying disaggregation to data center memory. By giving the compute nodes a hardware-managed DRAM cache, we can deliver a remote memory system that has the intuitive abstractions provided by virtual memory-managed remote memory systems but has the low overhead of explicit RDMA systems.

9.1 Dissertation Summary and Contributions

We began this dissertation by introducing and categorizing various existing remote memory systems. We compared the strengths and weaknesses of explicitly-accessed systems like Infiniband, virtual memory-managed systems like Infiniswap and Remote Regions, and fully hardware-managed systems like CC-NUMA and COMA. We then discussed techniques used in the design of existing DRAM caches and how they could be applied to a DRAM cache for remote memory. Next, we introduced the tools we built to make it easier to design RTL models and simulate them on FPGAs in a cycle-accurate way. Then, we discussed the design of the remote memory system, starting with the memory blade and RDMA controller and continuing on to the DRAM cache controller and automatic prefetcher. Finally, we demonstrated the performance of the DRAM cache system on a typical interactive workload and a typical batch workload and compared it to the performance of a system using only local DRAM. The design and evaluation of this remote memory system constitute the main contributions of this work, which are summarized below.

- **Survey of existing remote memory implementations** - In this work, we surveyed various existing remote memory systems such as Infiniband, UPC, Infiniswap, Remote Regions, CC-NUMA, and COMA. Most of the systems we surveyed used virtual memory-managed caching. We showed that these systems have a major downside in the form of high overhead and suggested a hardware-managed DRAM cache as a way of reaping the benefits of caching without paying the performance overhead. We then looked at existing DRAM cache designs, which use on-package high bandwidth DRAM to cache off-page DRAM, and considered which techniques used for those systems would be applicable to a DRAM cache for remote memory.

- **RTL implementation of remote memory system** - The primary contribution of this work is the implementation of RTL generators for the various parts of the remote memory system, including memory blade controller, remote memory client controller, and DRAM cache controller. Each of these generators is parameterizable, with configuration options controlling the number of banks, number of MSHRs, cache size, cache associativity, cache line size, and the presence of features such as an SRAM tag cache. By making the RTL generator configurable in this way, we could perform a design space exploration to find the optimal settings.

- **Realistic evaluation of remote memory system** - Using the RTL generators, we performed microbenchmarks to show how virtual memory-managed systems have unacceptable overhead and how the hardware-managed DRAM cache system removes that overhead. We also performed a design space exploration on the DRAM cache to show that, in the presence of automatic prefetching, small cache lines and SRAM metadata caching were the optimal configuration. We then ran a representative interactive workload, Memcached, and showed how the DRAM cache could serve as an improvement over a software-based proxy for scaling up to multiple nodes. We also ran a representative batch workload, a graph algorithm, and showed how DRAM caching could offer easy scaling at the cost of a moderate performance hit.

9.2 Future Work

In this section, we will discuss the aspects of remote memory that weren't covered in this dissertation and are left as future work.

- **Multiple Applications** - In this work, we analyzed the performance of individual applications running on the remote memory system. However, we have not explored how different applications running simultaneously on the remote memory system might interact with each other and affect each others' performance. In future work, we will analyze the performance of co-hosted applications. We are especially interested in how batch and interactive workloads could share the remote memory. We will also consider

how the design of the DRAM cache or memory blade could be changed to provide resource guarantees to applications.

- **Improved Prefetcher** - The automatic prefetcher designed for this work was relatively simple. It could only predict and prefetch completely linear memory access patterns. This worked quite well for SAXPY, but not as well in an application like friends-of-friends, which had a more irregular access pattern. It is possible that, with a more sophisticated predictor, the performance of the friends-of-friends benchmark using the DRAM cache could be made to match performance using local DRAM. One example of a prefetcher design to explore would be one that remembers previous non-consecutive access patterns and prefetches them when it sees the same pattern again. This might be helpful for algorithms that re-execute certain code paths after doing other work.

- **Allocation** - In Chapter 2, we saw that remote memory systems have essentially two choices in how they handle allocation. Allocation decisions can either be centralized to a single manager or distributed across multiple nodes. In our evaluation, we largely avoided the question of allocation by only having one process acting as the remote memory client. This way, the process could take up the entire remote memory address space by itself. In future work, we will use benchmarks involving multiple compute nodes to explore different methods of allocating remote memory. We are especially interested in how allocation decisions affect load balancing among multiple memory blades.

- **Coherence / Writable Shared Memory** - Because all of our benchmarks involved a single process acting as the client, we did not have to deal with the issue of coherence. We purposefully did not design the DRAM cache to perform hardware-managed coherence across nodes because of the complexity that would involve. The DRAM cache hardware does have an explicit flush command which could be used to implement software coherence. It would be interesting to see how well software coherence could scale in such a system.

- **Fault Tolerance** - If memory is distributed to remote nodes, it is inevitable that a memory blade will go down or become inaccessible to the client. In a realistic remote memory system, the client-side software and hardware must be designed to be resilient to these sorts of failures. The client must be able to detect when a memory blade becomes inaccessible and either recompute or recover the data that was stored on it. Fault tolerance was outside the scope of this work, but we hope to explore this issue in future. This will likely require additions to FireSim to simulate node or network failures.

- **Cost/Benefit Analysis** - One major factor that we have not considered in our evaluation is cost. A system could deliver performance improvements but still not be worth

a production deployment if the capital cost is too high. It is difficult to perform such an analysis at the moment because the silicon photonic technology our design relies on is not fully mature. As this technology develops more in the future, it will be possible to perform a proper cost/benefit analysis.

Bibliography

[1] A. Agarwal et al. "The MIT Alewife machine: architecture and performance". In: *Proceedings 22nd Annual International Symposium on Computer Architecture*. June 1995, pp. 2–13. DOI: 10.1109/ISCA.1995.524544.

[2] Marcos K. Aguilera et al. "Remote regions: a simple abstraction for remote memory". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 775–787. ISBN: 978-1-931971-44-7. URL: https://www.usenix.org/conference/atc18/presentation/aguilera.

[3] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[4] J. Bachrach et al. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012*. June 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[5] C. Celio. *CCBench*. https://github.com/ucb-bar/ccbench.

[6] Christopher Celio, David A. Patterson, and Krste Asanović. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Tech. rep. UCB/EECS-2015-167. EECS Department, University of California, Berkeley, June 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html.

[7] UPC Consortium, Dan Bonachea, and Gary Funck. "UPC Language and Library Specifications, Version 1.3". In: (Nov. 2013). DOI: 10.2172/1134233.

[8] Henry Cook. "Productive Design of Extensible On-Chip Memory Hierarchies". PhD thesis. EECS Department, University of California, Berkeley, May 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-89.html.

[9] B. Fitzpatrick. *MemCached*. https://memcached.org/.

[10] S. Frank, H. Burkhardt, and J. Rothnie. "The KSR 1: bridging the gap between shared memory and MPPs". In: *Digest of Papers. Compcon Spring*. Feb. 1993, pp. 285–294. DOI: 10.1109/CMPCON.1993.289682.

[11] Juncheng Gu et al. "Efficient Memory Disaggregation with Infiniswap". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 649–667. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[12] N. Gulur et al. "Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2014, pp. 38–50. DOI: 10.1109/MICRO.2014.36.

[13] W. Terpstra H. Cook. *SiFive TileLink Specification*. URL: https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13%5C_tilelink%5C_spec%5C_1.8.1.pdf.

[14] E. Hagersten, A. Landin, and S. Haridi. "DDM-a cache-only memory architecture". In: *Computer* 25.9 (Sept. 1992), pp. 44–54. ISSN: 0018-9162. DOI: 10.1109/2.156381.

[15] A. Izraelevitz et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2017, pp. 209–216. DOI: 10.1109/ICCAD.2017.8203780.

[16] 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 '13. Tel-Aviv, Israel: Association for Computing Machinery, 2013, pp. 404–415. ISBN: 9781450320795. DOI: 10.1145/2485922.2485957. URL: https://doi.org/10.1145/2485922.2485957.

[17] Svilen Kanev et al. "Profiling a Warehouse-Scale Computer". In: *SIGARCH Comput. Archit. News* 43.3S (June 2015), pp. 158–169. ISSN: 0163-5964. DOI: 10.1145/2872887.2750392. URL: https://doi.org/10.1145/2872887.2750392.

[18] S. Karandikar et al. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud". In: (June 2018), pp. 29–42. DOI: 10.1109/ISCA.2018.00014.

[19] Sagar Karandikar et al. "FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 715–731. ISBN: 9781450371025. DOI: 10.1145/3373376.3378455. URL: https://doi.org/10.1145/3373376.3378455.

[20] K. Koh et al. "Disaggregated Cloud Memory with Elastic Block Management". In: *IEEE Transactions on Computers* 68.1 (Jan. 2019), pp. 39–52. ISSN: 2326-3814. DOI: 10.1109/TC.2018.2851565.

[21] James Laudon and Daniel Lenoski. "The SGI Origin: A ccNUMA Highly Scalable Server". In: *SIGARCH Comput. Archit. News* 25.2 (May 1997), pp. 241–251. ISSN: 0163-5964. DOI: 10.1145/384286.264206. URL: http://doi.acm.org/10.1145/384286.264206.

[22] Yunsup Lee et al. *The Hwacha Microarchitecture Manual, Version 3.8.1*. Tech. rep. UCB/EECS-2015-263. EECS Department, University of California, Berkeley, Dec. 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-263.html.

[23] D. Lenoski et al. "The Stanford Dash multiprocessor". In: *Computer* 25.3 (Mar. 1992), pp. 63–79. ISSN: 0018-9162. DOI: 10.1109/2.121510.

[24] J. Leverich. *Mutilate*. https://github.com/leverich/mutilate.

[25] Kevin Lim et al. "Disaggregated Memory for Expansion and Sharing in Blade Servers". In: *SIGARCH Comput. Archit. News* 37.3 (June 2009), pp. 267–278. ISSN: 0163-5964. DOI: 10.1145/1555815.1555789. URL: http://doi.acm.org/10.1145/1555815.1555789.

[26] G. H. Loh and M. D. Hill. "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches". In: *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2011, pp. 454–464.

[27] L. Yang M. Rajashekhar. *Twemproxy*. https://github.com/twitter/twemproxy.

[28] Jim Mcauley and Jure Leskovec. "Learning to discover social circles in ego networks". In: *NIPS* 1 (Jan. 2012), pp. 539–547.

[29] J. C. McCallum. *Memory Prices (1957-2019)*. https://jcmit.net/memoryprice.htm.

[30] Mihir Nanavati, Jake Wires, and Andrew Warfield. "Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 17–33. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/nanavati.

[31] Stanko Novakovic et al. "Scale-out NUMA". In: *SIGPLAN Not.* 49.4 (Feb. 2014), pp. 3–18. ISSN: 0362-1340. DOI: 10.1145/2644865.2541965. URL: https://doi.org/10.1145/2644865.2541965.

[32] M. K. Qureshi and G. H. Loh. "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2012, pp. 235–246. DOI: 10.1109/MICRO.2012.30.

[33] Charles Reiss et al. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: Association for Computing Machinery, 2012. ISBN: 9781450317610. DOI: 10.1145/2391229.2391236. URL: https://doi.org/10.1145/2391229.2391236.

[34] S. Sanfilippo. *Redis*. https://redis.io/.

[35] David Schor. *Coffee Lake - Microarchitectures - Intel*. URL: https://en.wikichip.org/wiki/intel/microarchitectures/coffee%7B%5C_%7Dlake.

[36] Yizhou Shan et al. "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 69–87. ISBN: 978-1-931971-47-8. URL: https://www.usenix.org/conference/osdi18/presentation/shan.

[37] J. Sim et al. "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2012, pp. 247–257. DOI: 10.1109/MICRO.2012.31.

[38] J. Sim et al. "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2012, pp. 247–257. DOI: 10.1109/MICRO.2012.31.

[39] P. Stenstrom, T. Joe, and A. Gupta. "Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures". In: *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*. May 1992, pp. 80–91. DOI: 10.1109/ISCA.1992.753306.

[40] Z. G. Vranesic et al. "Hector: a hierarchically structured shared-memory multiprocessor". In: *Computer* 24.1 (Jan. 1991), pp. 72–79. ISSN: 0018-9162. DOI: 10.1109/2.67196.

[41] S. Yu and P. Chen. "Emerging Memory Technologies: Recent Trends and Prospects". In: *IEEE Solid-State Circuits Magazine* 8.2 (2016), pp. 43–56.

[42] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, p. 10.