ABSTRACT

Title of dissertation: PERFORMANCE EXPLORATION OF THE HYBRID MEMORY CUBE

Paul Rosenfeld, Doctor of Philosophy, 2014

Dissertation directed by: Bruce Jacob
Department of Electrical Engineering

The Hybrid Memory Cube (HMC) is an emerging main memory technology that leverages advances in 3D fabrication techniques to create a memory device with several DRAM dies stacked on top of a CMOS logic layer. The logic layer at the base of each stack contains several DRAM memory controllers that communicate with the host processor over high speed serial links using an abstracted packet interface. Each memory controller is connected to several memory banks in the DRAM stack with Through-Silicon Vias (TSVs), which are metal connections that extend vertically through each chip in the die stack. Since the TSVs form a dense interconnect with short path lengths, the data bus between the controller and memory banks can be operated at higher throughput and lower energy per bit compared to traditional Double Data Rate (DDRx) memories, which uses many long and parallel wires on the motherboard to communicate with the memory controller located on the CPU die. The TSV connections combined with the presence of multiple memory controllers near the memory arrays form a device that exposes significant memory-level parallelism and is capable of delivering an order of magnitude more bandwidth

than current DDRx solutions.

While the architecture of this type of device is still nascent, we present several parameter sweeps to highlight the performance characteristics and trade-offs in the HMC architecture. In the first part of this dissertation, we attempt to understand and optimize the architecture of a single HMC device that is not connected to any other HMCs. We begin by quantifying the impact of a packetized high-speed serial interface on the performance of the memory system and how it differs from current generation DDRx memories. Next, we perform a sensitivity analysis to gain insight into how various queue sizes, interconnect parameters, and DRAM timings affect the overall performance of the memory system. Then, we analyze several different cube configurations that are resource-constrained to illustrate the trade-offs in choosing the number of memory controllers, DRAM dies, and memory banks in the system. Finally, we use a full system simulation environment running multi-threaded workloads on top of an unmodified Linux kernel to compare the performance of HMC against DDRx and "ideal" memory systems. We conclude that today's CPU protocols such as coherent caches pose a problem for a high-throughput memory system such as the HMC. After removing the bottleneck, however, we see that memory intensive workloads can benefit significantly from the HMC's high bandwidth.

In addition to being used as a single HMC device attached to a CPU socket, the HMC allows two or more devices to be "chained" together to form a diverse set of topologies with unique performance characteristics. Since each HMC regenerates the high speed signal on its links, in theory any number of cubes can be connected together to extend the capacity of the memory system. There are, however, practical limits on the number of cubes and types of topologies that can be implemented.

In the second part of this work, we describe the challenges and performance impacts of chaining multiple HMC cubes together. We implement several cube topologies of two, four, and eight cubes and apply a number of different routing heuristics of varying complexity. We discuss the effects of the topology on the overall performance of the memory system and the practical limits of chaining. Finally, we quantify the impact of chaining on the execution of workloads using full-system simulation and show that chaining overheads are low enough for it to be a viable avenue to extend memory capacity.

PERFORMANCE EXPLORATION OF THE HYBRID MEMORY CUBE

by

Paul Rosenfeld

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Advisory Committee:
Professor Bruce Jacob, Chair/Advisor
Professor Manoj Franklin
Professor Gang Qu
Professor Donald Yeung
Professor Jeffrey Hollingsworth, Dean's Representative

To my wonderful parents, Simon and Olga Rosenfeld

Without their support, none of this would have been possible.

Acknowledgments

First and foremost, I would like to acknowledge my family who has done everything in their power to support me in both my academic and personal life. My father's brilliance and boundless curiosity, my mother's encouragement and ability to ward off any oncoming panic attack with a few words, and my brother's unique perspective and guidance have all helped me focus on what is important. Their support underlies everything that I do, and I am thankful to have such wonderful people in my life.

I owe a special thanks to my girlfriend, Ivy Liu, who has had to endure a disproportionate number of computer references over the years. I imagine that she will forever think "memory fragmentation problem" when looking at a suboptimal seating arrangement (and for that, I apologize). She has helped me through several graduate school existential crises and has always been there for me—for that I am grateful.

Thanks to Elliott Cooper-Balis for being a friend and sounding board for ideas throughout my graduate school career. I wish all the best to the DRAM Ninjas past and present (Mu-Tien Chang, Ishwar Bhati, Jim Stevens, Paul Tschirhart). Avadh Patel who contributed so much of his time and effort furthering the open source and academic research communities and enabling our group's work. Arun Rodrigues for pulling us into the world of supercomputers. Finally, I would like to thank my advisor, Dr. Bruce Jacob, for his leadership and for always being our biggest proponent.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

BOB	Buffer-on-Board (memory system)
CAS	Column Access Strobe
CMP	Chip Multi-Processor
DRAM	Dynamic Random Access Memory
SDRAM	Synchronous Dynamic Random Access Memory
DDR	Double Data Rate
DIMM	Dual Inline Memory Module
FBDIMM	Full Buffered DIMM (memory system)
HMC	Hybrid Memory Cube
LRDIMM	Load Reduced DIMM (memory system)
NUMA	Non-Uniform Memory Access
RAS	Row Access Strobe
RRQ	Read Return Queue
TSV	Through-Silicon Via(s)

Chapter 1

Introduction

1.1 Status Quo: Current Generation Memory Systems

The original "Memory Wall" [1] paper was published well over a decade ago and the authors expressed horror at the idea that a memory access could take "tens or hundreds of" CPU cycles in a decade's time. Soon after their paper was published, synchronous DRAM began its slow march toward ubiquity after its standardization by JEDEC in 1993. What is remarkable about this situation is that while researchers have been sounding the alarm about the memory bottleneck for nearly two decades, today's DDRx memory systems look largely identical to the SDRAM systems from two decades ago.

With today's multi-core processors containing aggressive pipelines, superscalar execution, and out of order scheduling, the demands on the memory system are more stringent than ever. There are three major problems that today's systems encounter:

- Memory bandwidth per core is insufficient to meet the demands of modern chip multiprocessors

- Memory capacity per core is insufficient to meet the needs of server and high performance systems

- Memory power consumption is beginning to dominate large systems (i.e., high performance computing, data centers)[2][3]

1.2 Synchronous DRAM

The original SDRAM standard was adopted by JEDEC in 1993 and the vast majority of today's computers are still using updated variants of the original SDRAM: DDR1, DDR2, or DDR3. Although these subsequent improvements to the original standard allowed the devices to achieve higher bandwidth through signaling and timing improvements (such as the inclusion of On Die Termination, DLLs, doubling the data clock rate, etc.), the underlying architecture has barely changed since the original SDRAM standard.

A modern DDRx memory system consists of a memory controller that issues commands to a set of DRAM devices that are soldered to a ***Dual Inline Memory Module*** (DIMM) that is plugged into the motherboard. Each DIMM is made up of one or more ***ranks*** that are comprised of several DRAM devices connected to a common set of control and address lines (i.e., all of the devices in a rank operate in lockstep to perform the same operation on their local memory arrays). Within each device, there are multiple ***banks*** of memory each of which contains the circuitry required to decode an address and sense data from the DRAM array. A DDR3 device has eight banks per rank. The bank is the smallest independent unit of memory operations: commands destined for different banks can execute completely in parallel with respect to one another. A schematic image of the DDRx memory system can be seen in figure 1.1.

In order to reduce the number of address pins required, DDRx addresses are split into separate row and column addresses. The DRAM array inside of each bank is subdivided into rows of bits which are connected to a wordline driver. In the first phase of a data access, a row activation command (RAS) causes all of the devices in a particular bank to activate a wordline that contains an entire row of bits. Sense amplifiers detect the value of each DRAM cell and store the values in a row buffer. After the data has been stored in the row buffer, a column access command (CAS_W/CAS) drives data into or out of the DRAM array for a write or a read, respectively. Before a different row can be activated, a precharge command (PRE) must be issued to ready the sense amplifiers to sense a new row. The DRAM protocol also allows for a row to be implicitly precharged after a column access without having to send an explicit precharge command in order to reduce command bus contention.

The DRAM devices are designed to be "dumb" in that the memory controller is responsible for keeping track of the state of each bank in the memory system and guaranteeing that all device timing constraints are met. The device has no ability to detect timing violations; the data will simply become corrupt if the memory controller issues a command at the wrong time. Additionally, the memory controller must schedule all transactions such that there are no collisions on the shared data bus. In addition to avoiding collisions on the bus, the controller must also account turnaround time when the direction of the direction of the shared data bus changes.

In order to keep the core memory clock low with respect to the data bus clock, the DDRx standard specifies a minimum "burst length" for transactions to

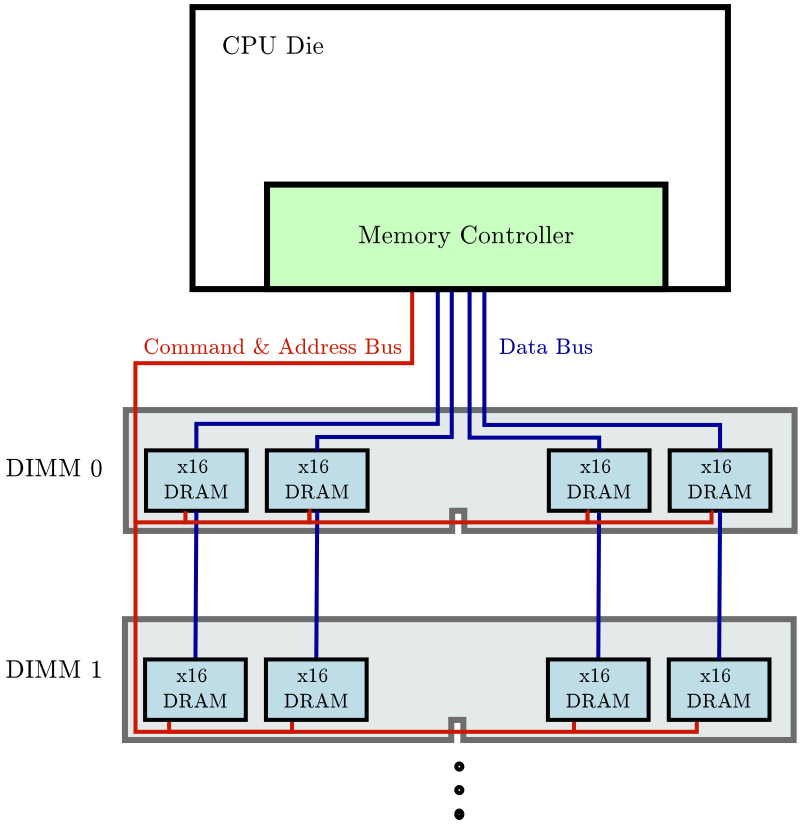


Figure 1.1: One channel of a traditional DDRx memory system. All of the ranks share common address, command, and data buses. The destination rank is chosen by a chip select line. The memory controller is responsible for ensuring all DRAM device timing constraints are met.

the DRAM array. Data is transferred to and from the memory controller over a wide 64-bit bus (or a 72-bit bus when ECC DRAM is used) in a series of "bursts". A DDR3 DIMM has a "burst length" of eight: a CAS triggers eight data transfers of 64 bits each for a total of 64 bytes. This corresponds to the size of a typical cache line fill (although a burst length of 4 can be requested for a 32 byte granularity). Data transfers happen on both the rising and falling edges of the I/O clock (hence the term Double Data Rate).

In non-embedded devices, a DIMM contains several individual DRAM devices that operate as a single logical device (see 1.1). The DRAM device width specifies the number of bits that the device will drive out onto the data bus during each data burst. For example, a DDR3 DIMM that uses x8 width DRAMs will contain eight devices per rank that are connected to the 64 bit data bus. Each CAS command selects a set of columns from the row and these bits are driven out of the I/O pins onto the shared data bus on each burst. That is, on each rising and falling edge of the I/O clock, each device will output a certain number of bits which are aggregated together onto the data bus.

Each device contains rows with 1K columns of 8 bits each for a total of 8 Kb per device [4]. On a row activation, all eight DRAM devices will activate 8 Kb for a total of 8 KB across the DIMM. This situation is even worse for a DIMM consisting of x4 parts where a full 16 KB is activated across the DIMM. Before a row is closed (precharged) any number of column activations can be sent without having to re-issue a RAS command. For a detailed description of DRAM protocol and operation see [5].

This multiplexed addressing scheme, however, results in an efficiency problem: a row access activates 8-16 KB while a CAS only drives 64 bytes of data into or out of the DRAM array. In a “close page” row buffer policy (where a row is closed immediately after every column activation), less than 1% of the activated bits are actually read or written. This is known as the “overfetch” problem [6] and results in one source of energy inefficiency in DDRx DRAM. Memory controllers may implement an “open page” row buffer policy which attempts to exploit spatial locality in the request stream to send multiple column access commands to an open row and thus increase the row utilization. However, this comes at the cost of logic complexity and queue space overhead in the controller. Even so, it is unlikely that a normal access pattern can exploit more than a few percent of the bits in an open row.

The wide parallel data bus in DDRx systems also creates a scaling problem. As the DRAM clock rates increase to try to keep pace with CPU bandwidth demand, the signal integrity becomes significantly degraded due to crosstalk and signal reflection. This problem is also exacerbated by the fact that electrical contact to the DIMMs is maintained by physical pressure from the DIMM slot contacts and not a permanent electrical connection such as with solder. As more DIMMs are added to the wide multidrop DDRx bus, the resulting capacitance increase and signaling problems create a situation where the data clock rate must be lowered in order to maintain signal integrity. This means that in order to achieve higher bandwidths, system designers typically reduce the number of DIMMs per channel and increase the clock rate. This, in turn, leads to a capacity problem since the number of CPU pins devoted to the memory system and the capacity of a single DIMM are not growing very quickly.

1.3 Currently Proposed Solutions

Recently, industry has come up with several solutions to try to address the various shortcomings of DDRx.

1.3.1 DDR4

Recently, JEDEC has completed the DDR4 standard that is slated to replace current DDR3 devices. DDR4 introduces advanced I/O technology such as Dynamic Bus Inversion, Pseudo Open Drain, along with new On Die Termination techniques to reduce bus power consumption and increase signal integrity. DDR4 devices will operate at 1.2V, resulting in a substantial energy savings over current DDR3 devices which run at 1.35V. The use of shorter rows in the DRAM array (512B per row compared to a typical 2048B per row in DDR3) results in a lower activation energy as well as a lower row cycle time.

DDR4 is expected to provide data rates from 1.6 GT/s all the way up to 3.2 GT/s (2x the data rate of DDR3-1600 [7][8]) in future parts. DDR4 devices will contain 16 banks organized into four bank groups (compared to DDR3's 8 independent banks) that result in a higher level of memory parallelism and higher throughput at the cost of increasing scheduling complexity [9]. Consecutive accesses to the same bank group will incur a longer access time than accesses to different bank groups. This means that consecutive requests must go to different bank groups in order to avoid idling the data bus and reducing throughput [10].

Since no systems yet support DDR4, there is no clarity on how many DDR4 DIMMs per channel will be supported. Intel's upcoming Haswell-E CPU is reported to feature four DDR4 channels each supporting only a single DIMM per channel [11] as discussed by previous sources [12]. However, other sources refer to DDR4 configurations with up to three DIMMs per channel at reduced data rates [13]. It is unclear from this document whether a three DIMM per channel configuration requires LRDIMM technology to be used (see section 1.3.2).

To overcome the potential capacity limitation due to single channel depth, the DDR4 standard contains TSV stacking extensions to create 3D stacked DIMMs containing up to 8 DRAM dies [8]. Currently, Samsung has announced Registered DIMMs with capacity of up to 32 GB per DIMM (and LRDIMMs with capacity of up to 128 GB per DIMM) [13].

1.3.2 Load Reduced DIMM

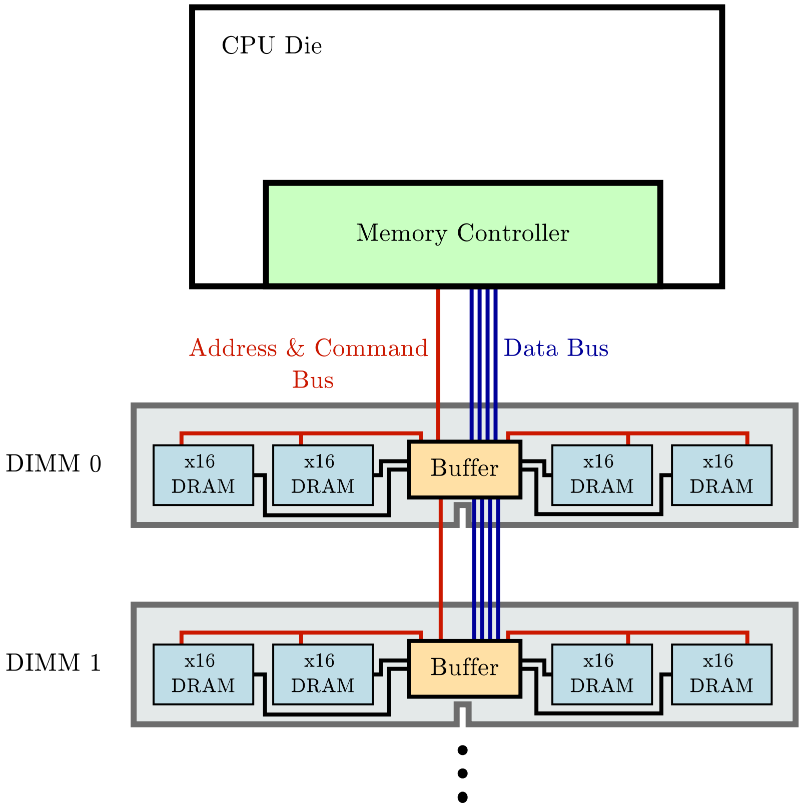


Figure 1.2: A Single LRDIMM channel. By placing a buffer chip on each DIMM to latch the control, address, and data lines, LRDIMM is able to reduce the loading on the DRAM bus. This enables faster clock speeds with higher channel depths as compared to a standard DDRx channel.

Load Reduced DIMM (LRDIMM) takes the approach of reducing the capacitive load on the memory buses by adding latching registers to the control, address, and data lines[1]. In a traditional unregistered or registered DIMM, each rank on a DIMM is its own electrical load on the bus. In an LRDIMM, however, multiple ranks are connected to the buffer chip that appears as a single electrical load on the bus. This means that a 32 GB quad rank LRDIMM results in a 4x load reduction compared to a normal Registered DIMM (RDIMM) [14]. The load reduction mitigates the capacity problem by allowing more DIMMs to be placed per channel than traditional DDRx systems while maintaining reasonably high clock speeds. For example, LRDIMM allows three DIMMs per channel at 1333 MT/s at 1.5 V whereas RDIMM only allows two [15]. Recently, Inphi and Samsung demonstrated a quad-socket server system containing 1.5 TB DRAM running at 1333MT/s (4 sockets x 4 channels per socket x 3 32GB DIMMs per channel) [16].

The LRDIMM load reduction technique can be utilized with DDR3 as well as upcoming DDR4 devices.

1.3.3 Fully Buffered DIMM

In 2007, JEDEC approved a new memory standard called Fully Buffered DIMM (FB-DIMM). FB-DIMM places a buffer chip called an Advanced Memory Buffer (AMB) on each memory module. The modules communicate with the memory controller using a high speed, full-duplex point-to-point link instead of a wide

[1] Note that this differs from a Registered DIMM that only latches the control and address signals, but not the data bus.

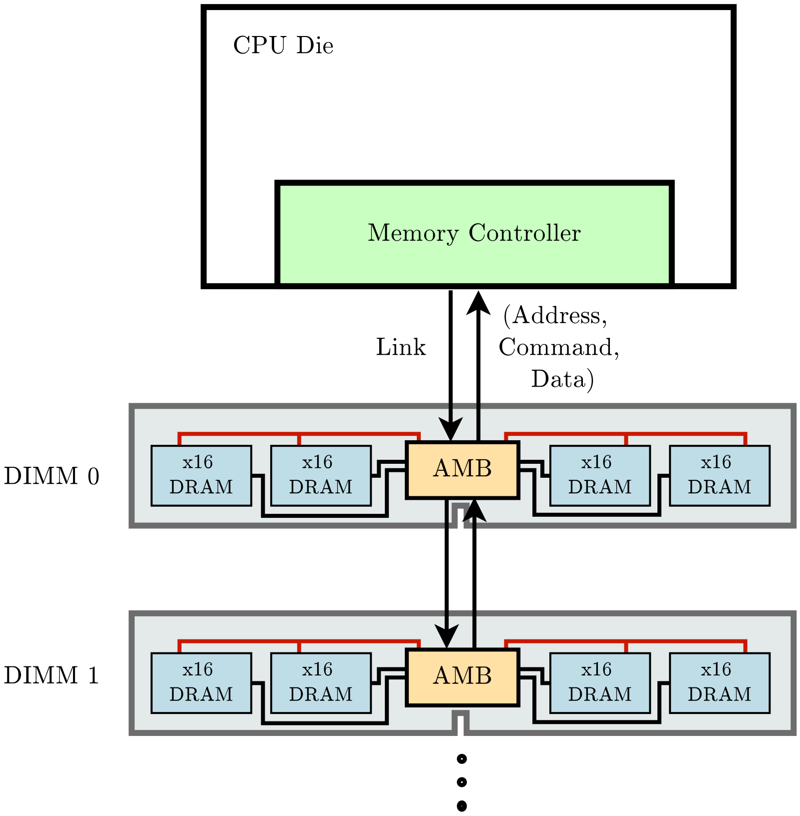


Figure 1.3: One FB-DIMM channel. The wide DDRx data bus is localized to a single DIMM. The Advanced Memory Buffer on each DIMM translates the high speed link protocol into DRAM protocol.

parallel bus. Since all connections between modules are point-to-point, each memory module must capture the data from the link and either process the request locally or forward the request to the next module in the chain.

By replacing the wide DRAM buses with high speed, point-to-point links, many more DIMMs can be placed in a channel while maintaining high data rate. Though FB-DIMM addressed the bandwidth and capacity problems of the memory system, it was never widely adopted. The power consumption of the AMB proved to be the biggest problem with FB-DIMM since it added a non-trivial power overhead to each DIMM. FB-DIMM allowed approximately 24x the number of DIMMs in the memory system [17] as compared to a DDR3 system while adding approximately 4 W of power overhead per DIMM [17][18] resulting in power overheads that could reach nearly 100 W. The power overhead of the memory was on par with CPU power consumption at the time.

In the end, FB-DIMM was abandoned by vendors and taken off industry road maps altogether.

1.3.4 Buffer-on-Board

Yet another approach to increasing capacity and bandwidth is the "Buffer-on-Board" (BOB) memory system. This type of memory system has been implemented by the major vendors (Intel, IBM, etc.) for their high end server systems. The BOB memory system is comprised of a master memory controller on the CPU die communicating with several slave memory controllers over high speed, full-duplex serial links. Whereas the CPU communicates with each slave controller using a packet-based protocol, the slave controllers communicate with commodity DDR3 DIMMs using a standard DDR3 memory protocol. To amortize the cost of each high speed link, each slave controller can control more than one DRAM channel.

By splitting off each slave controller and allowing it to act as a buffer between the wide DRAM channel and the CPU, the BOB memory system can achieve high bandwidth and large capacity. The capacity is increased because the serial interface requires far fewer CPU pins per channel as compared to a DDR3 channel. A large number of memory channels can use the same number of CPU pins as just a few

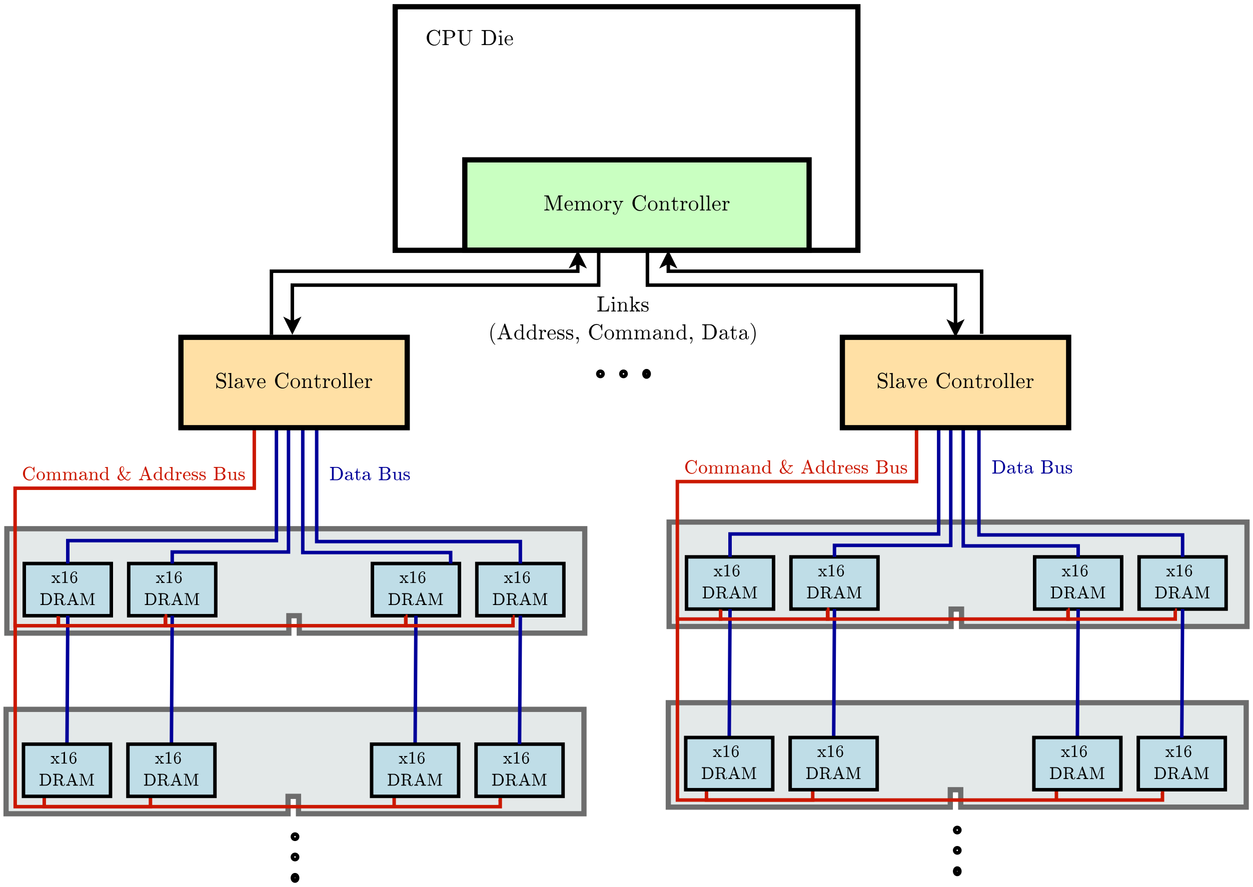


Figure 1.4: A Buffer-on-Board memory system. In the Buffer-on-Board memory system the main memory controller communicates with slave memory controllers over high speed links which then drive their own DDRx channels. By amortizing the cost of the slave memory controller over several DIMMs (instead of a single DIMM as in FB-DIMM), Buffer-on-Board is more cost and power efficient than FB-DIMM while maintaining its capacity and bandwidth.

DDR3 channels.

The high level of memory level parallelism enables better spread of memory requests to independent resources to shorten access time by avoiding conflicts. The independent, high speed, full-duplex links stream the data back to the CPU with minimal latency overhead. Overall, the system can achieve high bandwidth and high capacity with reasonable latency overheads. For example, [19] shows that a particular buffer on board configuration with 256 GB of memory connected to an eight core CMP is able to achieve sustained bandwidth of about 35G B/s with about a 110 ns latency (limit case simulations using random address streams show sustained bandwidths of around 60 GB/s).

While the performance of BOB systems is significantly higher than a regular DDRx system, a BOB system introduces a significant power penalty. Unlike the FB-DIMM memory system that requires a buffer for each memory module, the BOB system only requires a single buffer for one or more channels of memory. Although this cuts down on the number of buffers required in the system, there is still a significant power penalty for running the slave memory controllers. In addition to driving a standard DRAM bus, the slave controllers must send and receive data over high speed I/O links to the CPU. Since these links don't exist in a traditional DDRx memory system, they represent yet another power overhead. Finally, since the BOB memory system allows expanded capacity, the number of DIMMs in the memory system is higher, adding further to a typical system power budget.

1.4 Discussion of Proposed Solutions

All of the solutions discussed in the previous section address various shortcomings of the current DDRx memory system. The common theme among all of the proposed solutions is that they maintain the standard DDRx technology at their core. Most of these solutions use a standard DDRx DRAM device and improve it externally by adding extra circuitry (and DDR4, while redesigned, keeps most of the core DDRx technology intact). From a cost and business perspective, this is a lower risk approach: fabrication facilities do not need to be retooled to build exotic new memory technologies that may fail to gain widespread adoption. However, by keeping the core DDRx technology in place, the solutions only make incremental progress in increasing bandwidth, capacity, and energy efficiency. DDR4 doubles throughput and lowers power consumption, but may potentially suffer from capacity limitations due to the large CPU pin requirement and potential single channel depth. LRDIMM increases capacity while changing power and performance only nominally. FB-DIMM and Buffer on Board offer capacity and performance increase, but with a power penalty.

The longer term problem with these solutions is the widely predicted end of DRAM scaling. Current generation DDR devices are manufactured in a 20 nm process [20], but it is unclear how much further the DRAM technology process will be able to scale downwards while still being able to produce a device that can hold charge without having to be incessantly refreshed. Currently, technology node scaling allows for improvements in power consumption and density of the memory system. However, after the predicted end of DRAM scaling, there is no clear path to continue increasing density and lowering power.

In the long run, some form of 3D stacking will become necessary to keep pushing the performance, capacity, and power advancements in the main memory system. 3D stacking can be used to eliminate long, problematic wires (both going to the chip on the motherboard and the global wire length on a chip), to increase density by allowing more devices per package and per CPU pin, and to decrease power consumption through better electrical characteristics and shorter wires. In this dissertation, we will examine one such implementation of a 3D stacked DRAM system: the Hybrid Memory Cube.

Chapter 2

Hybrid Memory Cube Architecture

One of the recently proposed solutions to the bandwidth, capacity, and power problems of the main memory system is a new memory device called ***Hybrid Memory Cube*** (HMC) [21][22]. The HMC technology leverages advances in fabrication technology to create a 3D stack of dies that contains a CMOS logic layer with several DRAM dies stacked on top. While the idea of stacking memory on top of logic is not a new one, only recently has research into advanced fabrication techniques allowed for such a device to start becoming commercially viable.

2.1 HMC Architecture

The Hybrid Memory Cube proposes to combine several stacked DRAM dies on top of a CMOS logic layer to form a ***cube***. The term "hybrid" is used to describe the fact the device contains both DRAM dies as well as logic dies combined into a single stack. The dies in the 3D stack are connected through a dense interconnect of ***Through-Silicon Vias*** (TSVs), which are metal connections that extend vertically through the entire chip stack. A cross section of the dies and TSVs can be seen in figure 2.4.

To create these vertical connections, the device wafers are first thinned and then etched to form holes that completely penetrate the wafer. The holes are then

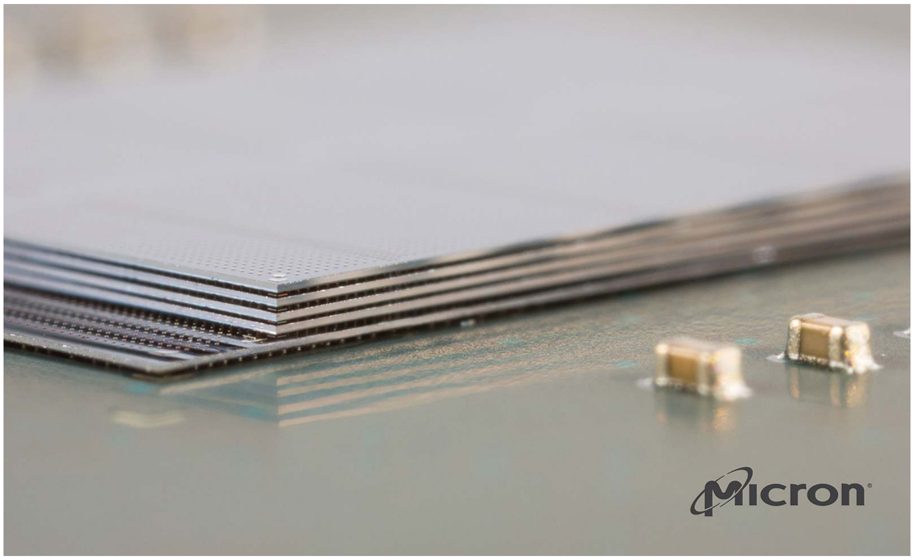


Figure 2.1: A closeup of an HMC stack.

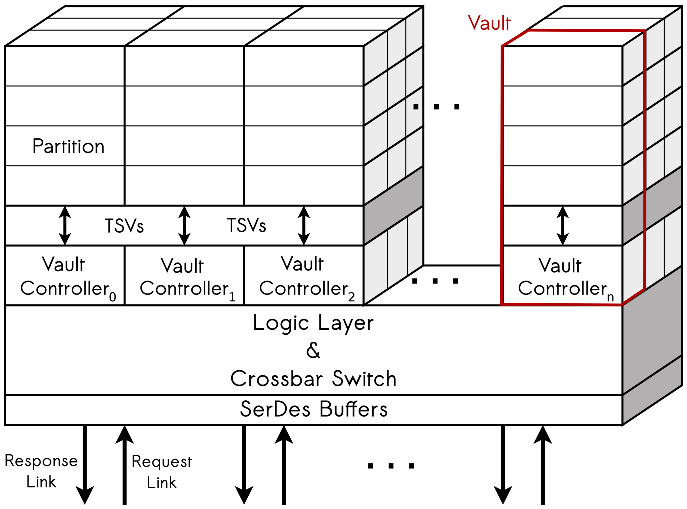


Figure 2.2: The architecture of an HMC cube. High speed serial links bring data into the cube which is routed to one of the vault controllers. Finally, the vault controller issues DRAM commands to the DRAM vaults. Read data flows from the vault controller through the interconnect and back out of the high speed links.

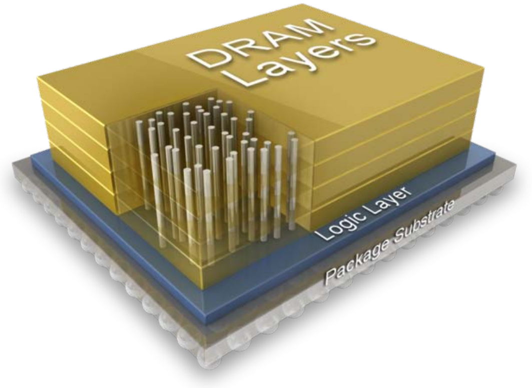


Figure 2.3: An artist's rendering of the HMC. Several DRAM dies are connected to a CMOS logic layer by vertical metal Through Silicon Vias (TSVs)

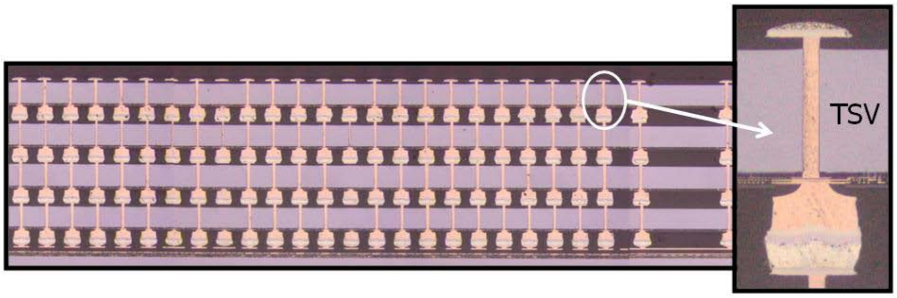


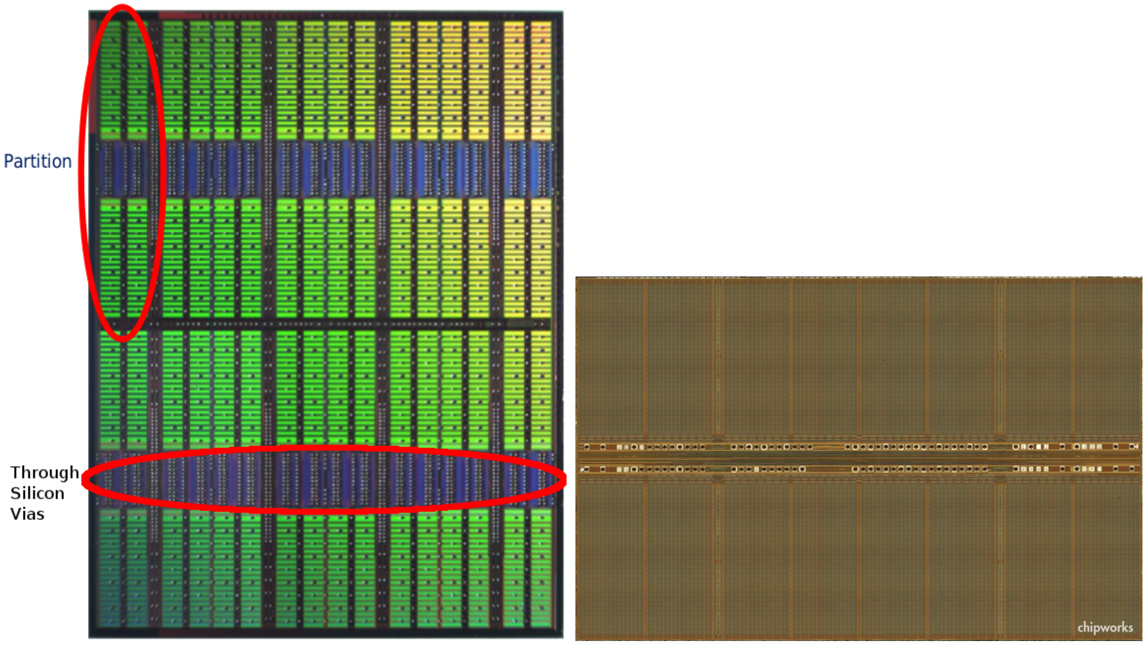
Figure 2.4: A cross section of an HMC device. A thinned wafer is perforated with vias that are filled with metal. Several wafers are bonded together to form a stack of devices connected with vertical metal connections (source: [23]).

filled with metal to form an electrical connection vertically through the entire wafer. Finally, several wafers are bonded together to form a 3D stacked device that can be packaged (see figure 2.1 for a close up of the HMC device before it is packaged).

The result is a permanent dense metal interconnect between the DRAM dies and the logic die that contains thousands of TSVs [24]. Unlike a typical DDRx DIMM which maintains electrical contact through the pressure of the pin slot, the TSV process forms a permanent metal connection between the dies. Because the TSVs provide a very short interconnect path between dies with lower capacitance than long PCB trace buses, data can be sent at a reasonably high data rate through the stack without having to use expensive and power hungry I/O drivers [25]. Furthermore, smaller I/O drivers and simplified routing allow a high interconnect density between the dies.

In order to increase the parallelism of the architecture, the dies are segmented vertically into ***vaults***. Each vault contains several ***partitions*** that each contain several banks. A single DRAM die within the stack contains several different partitions as shown in figure 2.5.

The base of each vault contains a ***vault controller*** which takes on the role of a traditional memory controller in that it sends DRAM-specific commands to the DRAM devices and keeps track of DRAM timing constraints. The vault controller communicates with the DRAM devices through the electrical connections provided by the TSVs. A vault is roughly the equivalent of a traditional DDRx channel since it contains a controller and several independent ranks (partitions) of memory on a common bi-directional data bus. However, unlike a traditional DDRx system, these



(a) Single HMC DRAM Die (source: [23]) (b) Samsung 2Gb DDR3 DRAM die (source: chipworks)

Figure 2.5: Comparison of HMC and DDRx DRAM dies. Each DRAM die in the HMC stack contains several partitions each with several banks. The areas between the DRAM devices are taken up by the TSVs while the DDR3 die is almost entire devoted to memory arrays.

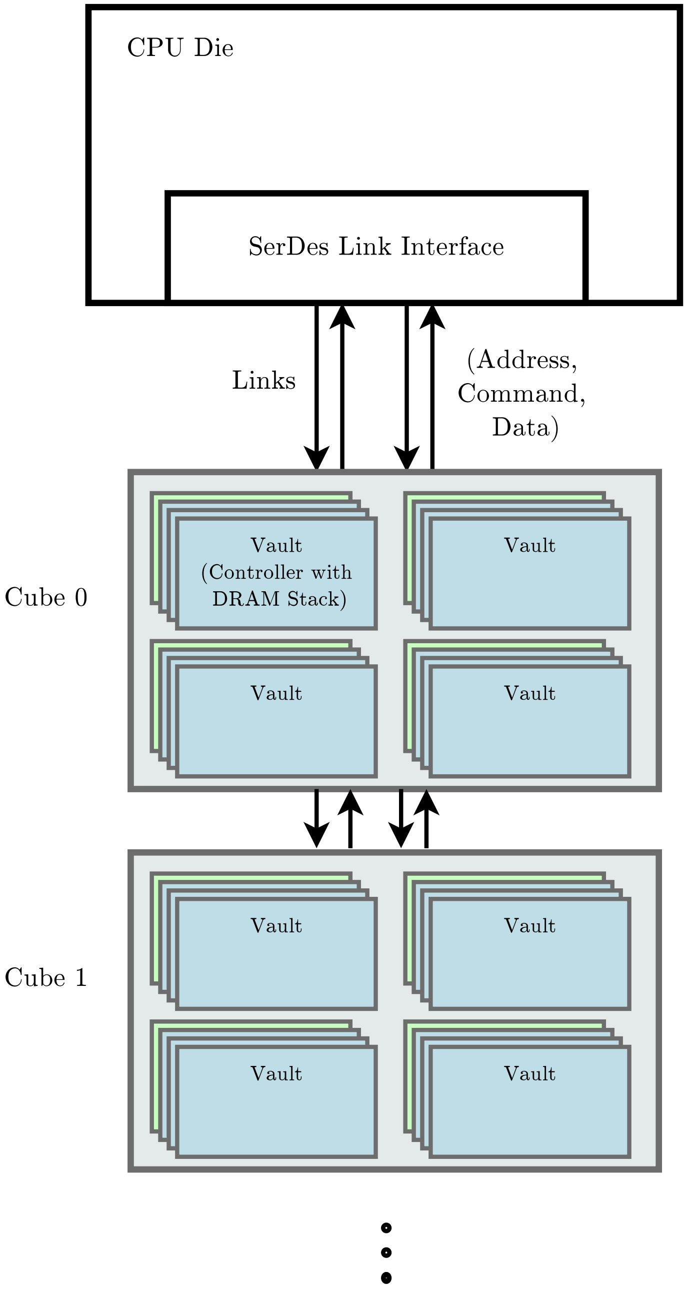


Figure 2.6: The architecture of an HMC memory system. The CPU communicates with cubes over a high speed packet interface. Cubes can connect to other cubes to form networks of memory.

connections are vastly shorter than the DDRx bus traces on a motherboard and have much better electrical properties. An illustration of the overall cube architecture can be seen in Figure 2.2.

In addition to containing several vault controllers, the logic layer interfaces with other cubes or hosts (e.g., CPUs) through a high speed link interface. Each link is comprised of several high speed lanes that typically run at several gigabits per second per lane. Although each individual lane is unidirectional and differentially signalled, each link is comprised of several lanes that run in both directions, making the link full-duplex. The link interface is responsible for serializing packets for transmission on the link's lanes by serializing them into the minimal transmission unit known as ***flits*** (as per the HMC specification, a flit is 16 bytes wide). When receiving packets, the link interface must de-serialize individual flits back into packets. Each packet contains metadata required for routing, error correction, and flow control.

The logic layer also provides a switch interconnect that connects the links to local vaults as well as to other links. The HMC specification states that any link must be able to access any local vault. Similarly, in order to support chaining of HMC cubes together, a packet from any link should be able to access pass-through links. There are two interconnect structures that are mentioned by the specification. The first is a full crossbar switch in which any link can transmit to any vault. The second is a segmented structure in which each link connects to a quadrant that services several local vaults. If a request arrives at a quadrant which does not contain the target vault, it can be forwarded to the appropriate quadrant that can service the request. However, details of the switching structure are not dictated by the HMC specification as the switching is completely hidden from the host controller. This is a major benefit to vendors as it allows them to optimize the switching structure without affecting the host protocol. This type of encapsulation is a feature of the HMC that will be described further in the next section.

2.2 Benefits of the HMC

The HMC architecture provides several key benefits over traditional DDRx memories that solve key problems plaguing current and future memory systems.

2.2.1 Capacity

One of the benefits of an HMC architecture is that it addresses the capacity and density problems of current DRAM technology. The capacitors inside of a DRAM die must maintain a minimum capacitance in order to be able to store charge long enough to avoid corruption or constant refreshing. It is difficult to shrink the DRAM cell size while keeping the same capacitance and thus improvements in DRAM density have slowed in recent years. Furthermore, it is unclear how much longer DRAM can continue to scale down to improve density. By leveraging Through-Silicon Vias, multiple DRAM dies can be stacked together (currently demonstrated parts have 4 dies, but 8 dies have been mentioned). With stacked dram dies, a single cube can contain a multiple of 4 or 8 times the storage in the same package footprint as a single DRAM device. In addition to decreasing footprint, the amount of capacity accessible per active CPU pin is increased as compared to a planar DRAM device. Note that the application of 3D stacking techniques to increase capacity is not unique to HMC. For example, as discussed in section 1.3.1, the DDR4 standard has 3D stacking extensions in order to help increase density without increasing pin count.

2.2.2 Parallelism and Aggregate Bandwidth

As previously mentioned, the TSVs are able to provide a high bandwidth connection between layers of the 3D stack. This high bandwidth is achieved through a combination of the density of the TSVs (there can be thousands of TSVs per cube) as well as the ability to transfer data at a high frequency. As TSVs are a short vertical path between dies, it is possible to transmit data at a high frequency without the signalling problems of many parallel long wires associated with PCB traces on a motherboard (as in DDRx systems). Furthermore, each cube has several high speed serialized links which achieve high bandwidth by using unidirectional differentially signalled lanes.

Since there are many independent vaults, each with one or more banks, there is a high level of parallelism inside of the cube. Each vault is roughly equivalent to a DDRx channel since it is comprised of a controller communicating with several independent DRAM devices sharing a data bus. With 16 or more vaults per cube, this means that each cube can support approximately an order of magnitude more parallelism within a single package. Furthermore, the vertical stacking of DRAM devices allows for a greater number of banks per package which is also beneficial to parallelism.

The architecture of the cube leverages many relatively slow DRAM devices put together in parallel to take advantage of the enormous bandwidth provided both by the TSVs that connect them to the controller as well as the high speed links that ultimately connect them to the CPU. Overall, depending on the particular cube configuration, tests on real hardware [26] and simulations in later chapters show that the HMC can deliver aggregate memory bandwidth of over 100 GB/s.

2.2.3 Energy Efficiency

By radically decreasing the length and capacitance of the electrical connections between the memory controller and the DRAM devices, the HMC is more energy efficient compared to DDRx memory devices. As previously mentioned, this also allows for the I/O driver circuitry to be simplified, making it more power efficient. Additionally, since much of the peripheral circuitry is moved into the logic layer, the power cost of this circuitry is amortized over a large number of DRAM devices, saving on overall power consumption. That is, each DRAM device is only responsible for reading the DRAM array and sending the data over a very short TSV bus, but unlike a traditional DDRx DRAM device, it does not need to communicate data all the way back to the CPU. Claims about energy efficiency range anywhere from 7x [27] to 10x [28] over current generation memory systems. Current estimates of HMC energy usage range from 10.48 pJ/bit [24] to 13.7 pJ/bit [21]. At peak utilization[1], it is estimated that DDR3 and LPDDR2 devices use approximately 70 pJ/bit and 40 pJ/bit, respectively [29]. Academic investigations also claim that a 3D stacked architecture like the HMC can be up to 15x more energy efficient than an LPDDR memory part [30].

2.2.4 Device Process Heterogeneity

Since a TSV process allows for heterogeneous dies to be stacked together, each die can be optimized for a specific purpose without having to sacrifice performance. The logic layer is optimized for switching and I/O while the DRAM dies are optimized for density and data retention. If these two dies were to co-exist in a single fabrication process, they would both suffer (i.e., DRAM built in a logic process cannot be dense; switching logic built in a DRAM process cannot switch at a high frequency). As a result of the stacking, each die achieves good performance and energy efficiency while retaining almost all of the benefits of being on the same die.

2.2.5 Interface Abstraction

The original SDRAM standard purposely created many generations of “dumb” memory devices; the memory controller was in full control of the memory devices and was responsible for ensuring all timing constraints were met. This enabled DRAM devices to contain a minimal amount of circuitry that wasn’t related to ma-

[1] Note that “peak utilization” represents the most energy efficient scenario; “typical utilization” has a higher energy per bit.

nipulating the DRAM array and driving the data on the data bus. While this was a rational design decision at the time, it had the effect of curtailing innovation in the memory system. Once the standard was written, nothing could be done to change the protocol. Any deviations from the standard DRAM protocol required the agreement of DRAM manufacturers, motherboard manufacturers, CPU manufacturers, etc. As modern CPUs began to incorporate the memory controller onto the CPU die for performance reasons, the problem became even worse: every processor model could only work with a single type of DRAM and would have to be intimately aware of every timing constraint of the memory system. These two factors together meant that any attempt at innovation in the memory system would usually take the form of adding external logic that was invisible to the host (ex: LRDIMM) or used commodity parts with a change to the host (ex: Buffer on board). Nothing could be done inside of the actual memory device itself.

Furthermore, the commodity economics of DRAM further stifled attempts at innovation. Even if improvements could be made inside of a DIMM that did not modify the DRAM protocol, they would largely be ignored as such improvements would add to the cost of a product in a commodity market where consumers make their decisions based largely on cost.

In the Hybrid Memory Cube, however, the memory device at the end of the communication link is no longer "dumb". That is, the CPU can communicate with the cube (or topology of cubes) over a general protocol that is then converted into device-specific commands within the vault controller. This allows for innovation in a number of different ways. The first improvement is that the DRAM timing inside of the cube can be changed without changing the CPU interface. Since the CPU has to simply operate at the read/write level and not at the DRAM protocol level, it no longer has to be intimately aware of every timing parameter associated with the memory device. This means advances in a DRAM can be integrated into the HMC without having to design an entirely new memory controller or CPU.

A second benefit of an abstract interface is that it allows any communication medium to be used as long as it is capable of delivering packets between a CPU and memory cube. Already researchers are thinking about how to replace electrical SerDes with high speed optical interconnects [31][3][32] to decrease power consumption.

Finally, an abstract interface provides a method of future-proofing the memory system. The authors of [3] point out that in the past, the CPU had to control hard disks at a very low level until the serial ATA (SATA) interface came along and pushed all of the control details into the disks themselves while only exposing an abstract interface to the CPU. This change enabled a painless transition from spinning disks to solid state disks by allowing vendors to conform to the high level SATA interface while managing the details of their own device. The same line of reasoning can be applied to the HMC: if the internal details of the memory technology are the responsibility of the cube and not the CPU, then it would be easy to change the underlying memory technology seamlessly if something came along to replace DRAM. There is an entire field of research into emerging memory technologies such as Spin-transfer Torque RAM (STT-RAM) and Phase Change Memory (PCM) that have a different set of trade-offs and access characteristics as compared to DRAM. An abstract protocol would enable them to migrate into an HMC-like device without exposing the change to the CPU. Furthermore, one could imagine heterogeneous memory stacks where DRAM dies and non-volatile memory dies co-exist in the same cube and are intelligently and transparently managed by the logic inside of an HMC.

2.2.6 Near-Memory Computation

Having a memory device with logic nearby opens the possibility of performing near-memory computations. Certain types of computations and operations could take operands from memory, perform some computation in the logic layer, and put the result back in memory or return it to the CPU. Although this idea is not new, there has never been such a convenient platform for implementing these operations since, as discussed in section 2.2.4, DRAM and logic can easily co-exist in an optimal way.

Already the HMC specification defines several commands that can be used for operations such as locking and read-modify-write that are best done near the memory instead of in the CPU. By executing these memory intensive operations near the memory, the HMC is able to reduce the amount of data that must be transferred back and forth between the memory and the processor.

Chapter 3

Related Work

The idea of stacking the memory hierarchy on top of the processor has been around for over a decade. Shortly after the "memory wall" paper was published, Kleiner et al. [33] proposed the idea of stacking the L2 cache (which was off-chip at the time) and DRAM on top of a RISC CPU and connecting the two chips using vias. They showed that such a stacked processor could perform instructions 25% faster. However, only recently has the fabrication technology progressed to the point where through silicon vias are actually feasible on a commercial scale. Buoyed by forward progress in TSV technology, there has been a recent surge of academic and industry research about possible applications of the technology.

3.1 DRAM on CPU Stacking

One of the biggest research directions for the utilization of 3D stacking has been the case where DRAM is stacked directly on top of the processor. This is a natural approach since the TSV process allows for heterogeneous dies to be stacked together (i.e., a CMOS logic CPU die with a DRAM die). This approach also provides the highest bandwidth and lowest latency path to the main memory, which leads to an effective way of bridging the performance gap between the processor and the memory.

3.2 System Level Studies

Liu, et al. [34] look at the advantages of stacking various memories (L2 cache or DRAM) directly on top of the CPU as well as the effects of deepening the cache hierarchy to include an L3 cache. Although they use a single core CPU model with dated benchmarks, they conclude that bringing the memory closer to the CPU via stacking can improve performance almost to the same level as having a perfect L2 cache.

In a similar but more recent study, Black, et al. [35] examine the performance and power implications of stacking either an SRAM or DRAM cache on top of a dual core CPU (unlike other studies, they do not assume the entire main memory will fit on top of the CPU). They show that using a 32MB stacked DRAM results in an average reduction of off-chip traffic by a factor of three, which translates to a 13% average reduction in memory access time and a 66% average reduction in bus power consumption. Their thermal modeling also shows that the temperature increases from stacking a single extra die are negligible and unlikely to impact the feasibility of stacking.

Kgil, et al. [36] show that stacking the main memory on top of a slow and simple multi-core CPU without an L2 cache can increase the network throughput for web server applications while achieving a 2-4x energy efficiency compared to more complex conventional CPUs. Their decision to use many simple cores with no L2 cache is motivated by their target workloads, which exhibit low locality and high thread level parallelism. The authors state that 3D stacking allows their chip to run

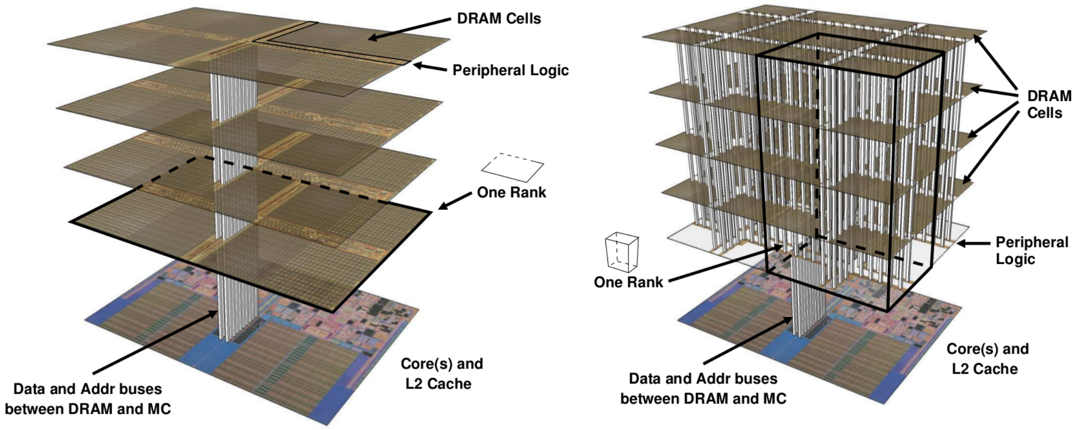


Figure 3.1: Two possible 3D rank organizations. Ranks are either split (a) horizontally ("3D") or (b) vertically ("true 3D"). The HMC model presented in this dissertation is more akin to the horizontal "3D" model in (a). (Image source: [38])

at a slower clock rate but still perform similar to a Pentium 4-like CPU at 1/10th of the power cost.

Loi, et al. [37] demonstrate that there is a 45% performance benefit to stacking memory on top of the CPU while maintaining a 64-bit bus width (same as conventional DRAM). Further utilizing the density of the TSVs to increase the memory bus width improves performance by up to 65% over conventional DRAM. Additionally, the authors show that even when taking into account the clock speed limitations imposed by thermal constraints of 3D stacked devices, these clock-limited stacked devices perform better than higher clocked 2D devices because of their ability to overcome the memory bottleneck with short, high density interconnections. There is, however, one shortcoming to this work as it only considers a 64 MB main memory size despite being a reasonably recent publication.

Loh [38] evaluates different organizations for 3D stacked DRAM. He shows that 3D organizations where each die contains a single rank (figure 3.1a) are not able to utilize the full bandwidth potential of the TSVs. The alternative approach of organizing the ranks in vertical slices (such that the banks are spread vertically among the dies; see figure 3.1b) results in the greatest performance increase. Stacking dies rank-wise yields a 35% performance increase (over a conventional off-chip DRAM configuration); further increasing the bus width to 64 bytes (so a cache line can be transferred in a single cycle) results in a 72% performance increase. Finally, going to a “true 3D” implementation with vertical ranks generates an impressive 117% performance increase. The work also contains a parameter sweep with varying numbers of memory controllers and ranks and shows that increasing the number of memory interfaces has better performance characteristics than increasing the number of ranks. The HMC model presented in this dissertation has an organization closer to figure 3.1a (i.e., the lower performing configuration): banks are contained within partitions located on a single DRAM die.

3.3 Low Level Studies

Facchini, et al. [25] develop a model for computing TSV power and use it to explore the energy-saving potential of stacking a DRAM die on top of a logic die in the context of mobile applications. They show that due to the improved electrical characteristics of the TSVs, simpler CMOS transceivers can be used to eliminate a significant portion of the bus power as compared to conventional DRAM.

CACTI-3DD [39] is an extension to CACTI-D to model the power, performance, and die area of 3D stacked DRAM. The authors also show, similarly to [38], that a “coarse-grain” stacking for each rank is unable to take full advantage of the TSV bandwidth, whereas a fine-grain “vertical rank” (where banks are spread vertically through the stack as opposed to horizontally on a single die) is able to achieve better performance and power characteristics.

Weis, et al. [30] explore the design space for how best to stack commodity DRAM dies for mobile applications with a focus on the internal structure (ex: number of banks per layer, number of data I/O TSVs). They find that a 1 Gbit 3D stack has 15x energy efficiency compared to LP-DDR devices when using a 64 Mbit tile size with x128 I/O.

3.4 Serially Attached Stacked DRAM

Udipi, et al. [3] examine how the use of emerging silicon photonics can be efficiently utilized to connect a CPU to an off-chip 3D stacked memory. Unlike much of the other related work in this area, their main memory is not stacked directly on top of the CPU. They propose an interface die that sits below the memory dies and converts the photonic packet interface into electrical signals and handles the low-level memory scheduling details. They examine various configurations of photonic stops in order to most effectively amortize the photonics power costs. Finally, they propose an “unscheduled” interface policy between the CPU memory controller and the DRAM to try to reduce complexity.

The work in [3] builds on their previous work [40] that proposes to alleviate the “overfetch” problem (i.e., bringing an enormous number of bits into the row buffers but only using a tiny fraction of them) by making DRAM rows much shorter. One of the biggest challenges they cite in this work is the lack of throughput between the smaller banks and the main memory controller. In their new work, the TSVs provide the low latency and high bandwidth path from the banks to the interface die, thereby eliminating the bottleneck. They are able to take advantage of the parallelism of a large number of banks through the usage of the TSVs.

Although the architecture presented in their work is very similar to the HMC (high speed links providing an abstract memory interface connected to an interface die with multiple memory controllers), their paper focuses more heavily on the photonics aspect of optimizing the architecture.

Kim, et al. [41] develop a “memory-centric network” of hybrid memory cubes that operate as both a storage and interconnect network for multiple CPUs. They attempt to maximize HMC storage capacity while lowering request latencies in the network through various routing techniques. This work focuses more on the interconnection aspects of the network of HMCs while we focus on the performance evaluation of a single cube and chains of cubes connected to a single host (and not multiple hosts). We also focus more heavily on understanding the internal dynamics of the memory cube as opposed to the traffic patterns between cubes.

Chapter 4

Methodology

4.1 HMC Simulator

All of the experiments performed are done using a C++ HMC simulator developed by our group. The cycle-based simulator models the entire HMC architecture including the high speed links, the link/vault interconnect, flow control, vault controller logic, and DRAM devices with their full set of timings. The initial version of the simulator was developed during a project that involved HMC modeling for Micron but was later modularized and generalized so that it can model any arbitrary 3D stacked memory device.

Although the internal logic and abstract protocol of the HMC has many practical advantages for the memory system (see 2.2.5), it also complicates academic HMC research. HMC vendors can hide the internals of the device behind the protocol and compete with one another by changing the internals of the device to deliver more performance with less power and cost. This type of competition is very difficult in a conventional DDRx system since all vendors are bound by the timing parameters and behaviors of the DDR specification and consumers make their purchases largely based on cost. However, this means that typically vendors will obscure the internal structure and parameters of the architecture. Parameters such as DRAM timings, clock frequencies, interconnect details, etc., become proprietary details that

Timing Parameter	**Value (cycles @ t_{CK}=1.25ns)**	**Time (ns)**	**Value (cycles @ t_{CK}=0.8ns)**
t_{RP}	11 cycles	13.75 ns	17 cycles
t_{CCD}	4 cycles	5 ns	6 cycles
t_{RCD}	11 cycles	13.75 ns	17 cycles
t_{CL}	11 cycles	13.75 ns	17 cycles
t_{WR}	12 cycles	15 ns	19 cycles
t_{RAS}	22 cycles	27.5 ns	34 cycles

Table 4.1: DRAM timing parameters used in simulations. Based on the parameters published in [41]

are closely guarded by vendors.

4.2 HMC Parameters

4.2.1 DRAM Timing Parameters

Being a memory device, one of the fundamental parameters in an HMC device is the DRAM timings. Unfortunately, as mentioned in the previous section, the DRAM timing inside of the HMC is proprietary information. The simulations performed here use the DRAM timings published in [41] as they are, to the best of our knowledge, the most comprehensive set of parameters that are currently published. Furthermore, they are more pessimistic with respect to the 45 nm DRAM timings presented in [30] and are similar to the timings mentioned in [42].

For simplicity, however, we assume that the vault clock is a multiple of the link reference clock which is defined as 125 MHz per the HMC specification [22]. To compute the vault clock, we take vault throughput of 10 GB/s [43] and divide it by the 32 TSV data lanes within each vault [24]. This yields a 2.5 Gb/s data rate, which, if we assume double data rate transmission, yields a 1.25 GHz TSV frequency ($t_{CK} = 0.8$ns). This frequency is precisely 10 times the link reference clock of 125 MHz. Table 4.1 shows the timing parameters used by [41], their equivalent time values, and our timing parameters based on a 0.8 ns clock period.

We note, however, that section 5.7.1 will show that small changes in DRAM timing parameters are unlikely to have a significant impact on the results presented here due to the large amount of memory-level parallelism in the HMC device.

4.2.2 Switch Interconnect

The role of the interconnect in the logic layer is to connect links to local vaults and to other links. In the single cube case, the switch interconnect only needs to connect links to vault vaults and vice versa. In the chained case, however, the switch also needs to connect links to other links to pass remote requests to a different cube.

Several presentations refer to the interconnect as a full crossbar [21], whereas the HMC specification refers to a switch structure where each link can service several local vaults with a lower latency than requests to non-local vaults. We will assume the switch interconnect is a crossbar switch due to the lack of details of the implementation of a different switch structure.

Since a flit is the smallest unit of data transmission, we assume that all of the data flows through the links and switches as individual flits until they arrive at the vault controller where they are re-assembled into transactions. Although the flits travel through the system as individual pieces of data, we enforce that flits from two transactions can never interleave when travelling between two endpoints (i.e., all of the flits in a transaction must complete between two points before the flits from another transaction can transmit).

4.2.3 Vault Controller

As transactions arrive at the vault controller from the switch, their flits are re-assembled into transactions and are placed in a command queue. The vault controller uses a First Ready First Come First Serve (FR-FCFS) [44] policy where ready requests can bypass stalled ones. While the controller will reorder requests, it will not reorder dependent requests ahead of one another.

Read responses are held in a "read return queue" as they wait to be transmitted on the switch back to a link. When a vault's read return queue fills up, the controller must stall the execution of all reads to avoid losing data.

We assume the vault controllers will use a closed page policy in which an implicit precharge command immediately follows the DRAM column access. That is, a row is immediately closed after data is written to or read from that row. This is in contrast to another row buffer policy called "open page" which leaves the DRAM row open by keeping its data in the row buffer and allows for subsequent read or write accesses to that row to be serviced from the buffer. For memory access patterns that exhibit high levels of temporal and spatial locality, leaving a DRAM row open can increase performance by amortizing a single row activation over many column accesses. On the other hand, open page row buffer models also impose a logic cost as the scheduling hardware is typically more complex: queues must be scanned to find transactions that can be scheduled early to an open row to make full use of the open row. Heuristics about when rows are likely to be reused must be carefully designed to avoid needlessly keeping rows open and incurring a power penalty. Furthermore, when an open page policy is used on workloads with little locality, delaying the precharge between accesses to different rows increases the latency of requests.

Server and HPC workloads typically exhibit little or no locality either due to the underlying algorithm (e.g., pointer chasing or sparse floating point computations) or the execution model (e.g., highly threaded server workloads). There is indication that the HMC's DRAM devices have been redesigned to have shorter rows [24] (256 bytes rather than 8-16 KB in a typical DDR3 device). The reduced row length helps to save power by alleviating the so-called "overfetch" problem where many bits are brought into the sense amplifiers but few are used. Shorter rows, however, reduce the probability of a row buffer hit, making open page mode impractical. Moreover, with the large number of banks in each cube, it is more efficient to utilize the high level memory-level parallelism to achieve high performance rather than to rely on locality which may or may not be present in a given memory access stream. For these reasons, we select a closed page row buffer policy for the simulations presented here.

4.3 Random Stream Methodology

In order to gain an understanding of the performance characteristics of various design choices, an exploration of the design space is required. Such a step through the design space involves changing many variables simultaneously and thus requires hundreds or potentially thousands of individual simulation runs. Running a full system simulation for each combination of variables is not feasible given that a full system simulation can take anywhere from several hours to several days. Therefore, to gain an initial understanding of how different variables affect performance, we perform random stream simulation for both single and multiple cube configurations.

In addition to having a much lower execution time than full-system simulations, random stream simulations have several useful properties that make them ideal for an initial design space exploration:

- **Zero Locality:** Random streams exhibit no spatial or temporal locality which represents the worst case performance scenario for modern processors as it renders caches ineffective for hiding memory latency. Furthermore, modern server applications execute many threads which substantially decrease the amount of locality in the memory access stream [45]. Similarly, many high performance computing applications utilize sparse data structures which require dereferencing multiple pointers to reach a particular data item (graphs, sparse matrices, etc.) Such "pointer chasing" algorithms generate a memory access stream that appears to be random. Therefore, a random stream is a good first order approximation for these types of applications as it represents a worst-case scenario where caches are ineffective and there is maximum load on the main memory system.

- **Address Mapping Agnostic:** Since each bit in a random address is equally likely to be zero or one, each request is equally likely to go to any memory bank/row in the system. Therefore, any address mapping scheme will have the same performance as any other address mapping scheme when driven by a random stream. Since the address mapping scheme will always tend to favor some particular access pattern, a random stream provides a way to look at the average performance of any address mapping stream. This eliminates the need to test different address mapping schemes and reduces the number of simulations required.

- **Determinism:** Since the random number generator can be seeded with a specific value, the same address stream can be generated to guarantee a fair comparison between different configurations. This is not always possible in a full system simulation where multiple factors can make the simulation non-deterministic.

- **Controllable Read/Write Ratio:** As we will later show in section 5.2.1, the performance of an HMC cube is dependent on the read/write ratio of the memory access stream. Unlike a particular application, a random stream generator can be tuned to produce memory accesses with deterministic read/write ratios.

The random stream simulations execute five million random transactions with a specific read/write ratio. Five million transactions allows the simulation to execute in steady state long enough to average out the transient start-up and tear-down portions of the simulation (i.e., waiting for the memory system to fill up and empty out). The driver program monitors the flow of requests to ensure that all transactions complete inside the HMC simulator before stopping the simulation.

To generate the stream, the driver generates a random 64-bit address and assigns it to be either a read or a write with probability equal to the specified read/write ratio. All random numbers are generated using the standard GNU C library routines (`rand()`, `randr()`, etc.). The random number generator seed for each simulation is set manually so that each set of configurations within a sweep being tested execute identical random streams (i.e., a set of simulations with varying parameters all have the same memory stream). Each set of configurations is executed three times with different seeds and the results are averaged to reduce the likelihood of a corner case result. In practice, however, we have observed very small standard deviations in performance metrics (bandwidth, latency) from different runs with different seeds.

The driver is configured to issue requests "as fast as possible": as soon as a cube's link is able to accept the transaction, the driver sends the transaction to the link. This mode of operation is designed to show the limit case of what happens when a device is under full load.

4.4 Full System Simulation Methodology

4.4.1 Choosing a Simulation Environment

Unfortunately, given the immense complexity and cost of chip fabrication, it is rarely possible for academic researchers to implement their ideas in actual hardware. Therefore, in order to show the benefits of a proposed architectural feature or new technology, most research relies heavily on simulation. Typically, new hardware features or devices strive to improve some specific aspect of the system such as increasing reliability, increasing performance, or decreasing power. By comparing the simulated performance of a system with and without the hardware feature, one hopes to accurately capture the impact of that feature on some target workload. In our case, we hope to see the impact of various memory system parameters on the execution time of programs.

One option for simulating a workload is to first create a memory trace that captures the addresses of all loads and stores in a program. This trace can then be replayed through the memory simulator and some metrics of performance can be extracted from the simulation. Trace-based simulation has the advantage of being fully deterministic: identical memory streams are executed through all the memory models. However, trace-based simulations are not "closed loop" in that without capturing the dependence information in the memory stream, it is not possible to have the memory model stall the CPU model. That is, if the data of WRITE B depends on the value of READ A, real CPU would stall until READ A is complete before sending WRITE B. In a trace-based model, however, these dependencies are typically not tracked and so it is assumed that READ A and WRITE B can simply execute through the memory model in parallel. Given the complexity of modern processors and workloads, we feel that a trace-based simulation cannot give an accurate picture of the performance impacts of a new memory device and therefore we do not consider any trace-based approaches.

In order to capture more realistic behavior, we consider CPU simulators that actually simulate the underlying processor ISA as a program is executing on simulated hardware. By connecting different memory models to the CPU simulator, we can capture the performance impact of the memory system on the program execution. Unlike the trace-based approach, this method is “closed loop” in that the memory performance affects the CPU performance and vice versa. That is, if the CPU issues many requests that swamp the memory system, the memory system may return responses more slowly (for example, due to conflicts), and thus the CPU must stall waiting on outstanding requests, which will result in a decrease in memory pressure. Capturing this feedback loop is essential in quantifying the performance impact of various memory technologies on workload performance. Given the goal of this work to understand the impact of the memory system on programs, it is clear that an ISA simulator is required.

Choosing an appropriate CPU simulator, however, can be challenging as there are several academic and commercial simulators available that vary widely in their features, fidelity levels, and target ISAs. As most HPC and server systems still use x86 CPUs, we limited our search to those simulators that support the x86 ISA. Targeting x86 also has some positive implications on tool chains and simulators:

binaries are easier to build when a cross compiler is not required and some simulators can offload simulated instructions to the hardware if the simulated ISA matches the host ISA.

Many simulators only offer a “system call emulation” mode where the operating system is emulated by simply emulating the system call interface. In most non-embedded environments, the user program can only access the hardware through the kernel via the system call interfaces. By emulating the system call interface, the CPU simulator monitors the user code’s accesses to hardware while bypassing the kernel completely. The user process can execute without ever being aware that it is running in a simulator with no actual devices.

A benefit of this approach is that it is much faster than simulating the full detail of the OS kernel: the user code is the only code that is being run in the CPU simulation. System call routines are typically implemented by remapping them to the underlying operating system of the host machine. For example, if a process opens a file handle to write a file, a system call emulation implementation can simply create a file descriptor in its own process and write the data to it. This behavior, while functionally correct, does not capture the hardware behavior of the underlying devices.

Since there is no kernel running in system call emulation mode, it cannot capture the full complexity of a modern operating system. The overhead of context switching, process scheduling, interrupts, and virtual memory are all lost in system call emulation mode. When trying to characterize the performance of a workload with a given memory system, it is important to capture the impact of virtual memory as it can have a noticeable impact on the memory access pattern seen at the physical memory. That is, since all modern operating systems utilize virtual memory for paging and protection, memory accesses that appear to be contiguous in the logical address space might be split at page boundaries by the operating system and placed into completely unrelated portions of physical memory. A system call emulation simulation cannot account for this important memory access behavior as it is the only executing process with no real operating system or virtual memory system underneath.

In addition to virtual memory, most modern devices make use of Direct Memory Access (DMA) to move data to and from hardware devices. Since system call emulation mode does not model any hardware except for the CPU and memory, it cannot capture the extra memory traffic generated by DMA devices. Similarly, memory traffic generated by the kernel itself cannot be captured in system call emulation mode. Although full system simulation is significantly slower than system call emulation simulation, we feel it is necessary to perform full system simulations to capture the full complexity of both the workload and the operating system it is running on. Therefore, we also limited ourselves to looking only for full system simulators.

In the end, we narrowed down the options to one of two simulators: the gem5 simulator [46] and MARSSx86 [47]. The gem5 simulator is a popular choice for academic computer architecture research because it is highly flexible. It supports a number of different ISAs including ALPHA, ARM, MIPS, x86, and SPARC. It also supports a number of different levels of CPU detail ranging from basic single-IPC

CPU	8 out-of-order x86 cores @ 3.2 GHz Issue Width: 4 128 Reorder Buffer Entries 48 Load Queue Entries 48 Store Queue Entries
L1 Cache	128 K L1-I / 128 K L1-D
L2 Cache	2 MB shared L2
Hardware Prefetch	Disabled
Operating System	Ubuntu Linux 11.04 Kernel 2.6.38

Table 4.2: MARSSx86 Configuration

models to fully pipelined, out of order models. Gem5 offers both a full-system mode that can boot an unmodified Linux kernel as well as a system call emulation mode. Unfortunately, at the time when we began this research, the x86 ISA was not fully functional in gem5 and we chose the much more stable x86 support in MARSSx86.

4.4.2 MARSSx86 Simulator

MARSSx86 is a cycle-based simulator that models an out-of-order, superscalar x86 multi-core CPU with a flexible cache hierarchy. It combines the emulation capabilities of QEMU with very detailed x86 timing models which allows it to boot an unmodified Linux operating system (see figure 4.1). Once a simulation begins, both user and kernel execution are simulated.

We augment the MARSSx86 simulator by replacing the built-in memory controller model with hooks to our own memory models. These include our HMC simulator, DRAMSim2 [48] to simulate the DDR3 baseline, and our “perfect memory” model that is described in section 4.4.3 to utilize our HMC simulator. We

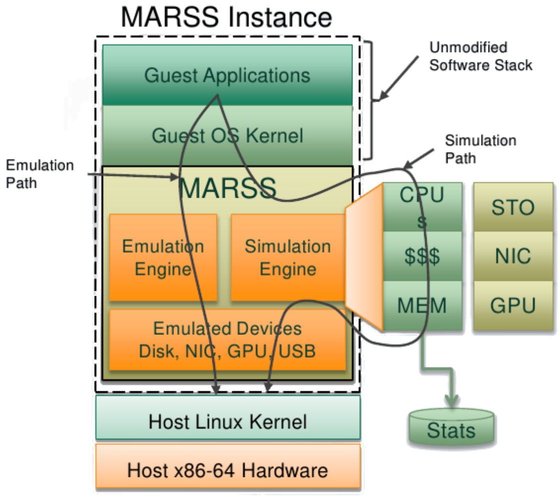


Figure 4.1: A block diagram of the MARSSx86 simulator. MARSSx86 combines QEMU emulated hardware with a CPU simulation path.

create identical software bindings to all three memory simulators so that they can utilize the same memory controller hooks within MARSSx86.

MARSSx86 is configured to simulate an eight core CPU running at 3.2 GHz and 4 GB of memory. A summary of the parameters for MARSSx86 can be found in table 4.2.

In order to reduce simulation time and to ignore uninteresting parts of the workload such as application initialization and shutdown, all workloads are annotated with a ”region of interest” hook. These extra function calls are inserted into the workload to start the simulation directly before the interesting computation begins and stop the simulation after it ends. The PARSEC benchmarks contain predefined regions of interest while the other workloads are annotated by hand. All of the workloads presented here are compiled with OpenMP support and run as eight threads inside of MARSSx86 (which simulates 8 cores).

In order to reduce the variability and non-determinism of full-system simulations, we utilize MARSSx86's checkpointing capability to create a snapshot of the running system state right at the start of the region of interest. After a simulation completes, any changes to the underlying disk image and snapshot are discarded to ensure that the original snapshot is preserved. This allows for several different simulations with different parameters to run starting from an identical system state.

Since each run simulates an identical portion of a program's execution, it is possible to directly compute the speedup or slowdown between two runs by comparing the simulated runtime of the region of interest.

4.4.3 Comparison Systems

In order to provide useful context of the performance of HMC we choose two comparison points that represent an upper and lower bound. The lower bound is a quad-channel DDR3-1600 system simulated in DRAMSim2 with timing parameters taken from the data sheet for Micron DDR3-1600 device MT41J256M4. The channels are configured in "unganged" mode to expose the maximum amount of memory-level parallelism. Configured in such a way, this DDR3-1600 system has a maximum theoretical peak bandwidth of 51.2 GB/s. It should be noted that although we refer to this as a "lower bound", this is quite an aggressive comparison point as only high end server CPUs have enough pins to implement a quad channel DDR3 memory system.

Additionally, we build a simple "perfect" memory model in order to put an upper bound on potential speedup of an application. This model only adds latency and has infinite bandwidth (i.e., any number of requests can be queued to the memory system per cycle and all are completed after a fixed latency). We add a latency of $t_{RCD} + t_{CL} + t_{Burst} = 19ns$ for a DDR3-1600 device, which represents the minimum time to open a row and stream data out of that row. Unlike real DRAM, this model implies that the memory system has infinite parallelism, no conflicts, and is perfectly deterministic. Although it is not possible to build such a memory in real life, it serves as a way to characterize how memory intensive a particular application is. That is, if a particular workload's execution time cannot be significantly decreased with the perfect model, the workload is not memory intensive.

If we compare the simple speedup of memory device B over another memory device A for a given workload and see only a small speedup, we may reach the conclusion that device B is only marginally better than device A. However, this may turn out to be the wrong conclusion if the workload only rarely accesses the main memory. If, instead of a simple speedup, we compute a slowdown from perfect for each memory system, we account for both the performance of the memory system but also the memory intensivity of the workload. In this case if memory device A and memory device B both have a small slowdown from perfect we can conclude that the workload is simply not able to stress the main memory enough to show a difference between the devices. If, however, both devices have a very large and similar slowdown from perfect, we can safely conclude that memory device A and memory device B are being adequately accessed by the workload and have similar performance.

Chapter 5

Single Cube Optimization

5.1 Motivation

The HMC architecture contains a large design space of parameters to be explored in order to optimize memory performance.

Within the DRAM stack, decisions must be made as to how to expose the proper level of memory parallelism to effectively utilize the available TSV and link bandwidth. That is, the links can be configured to provide tremendous throughput (terabits per second), but if the TSVs and memory storage elements are not able to utilize this bandwidth, then the throughput at the links is wasted. Some examples of choices inside of the cube include the number of independent vaults, the number of DRAM dies that should be stacked on top, and how many banks are required.

In this chapter, we attempt to characterize the performance trade-offs associated with different design parameters. First, we will try to understand how the serial full-duplex links change the HMC's performance as compared to a conventional memory system. This includes parameters such as link, TSV, and switch throughputs and queueing parameters. Next, to gain a general understanding of the design space, we simulate different HMC configurations being driven by a tunable random stream as described in 4.3. This includes a discussion of the relationship between the link and TSV bandwidth in section 5.2, a "constrained resource sweep" (section 5.5) where different HMC configurations of equal bandwidth and parallelism are compared. Then, we use full system simulation to illustrate the sensitivity of the overall cube performance to DRAM parameter changes. Finally, using the results from the random simulations as a guide, we proceed to characterize the full system performance of the HMC, as compared to DDR3 and "perfect" memory systems in section 5.6.

5.2 Link Bandwidth Optimization

5.2.1 Link Efficiency and Read/Write Sensitivity

In this section we attempt to understand the performance issues associated with the high-speed bi-directional links of the HMC memory system. Unlike a traditional DDRx system where the CPU communicates directly with the DRAM chips over a dedicated set of command, address, and data lines, the CPU communicates with the HMC using symmetric full-duplex serial links that transmit requests and data in packets. Similar to network traffic, each packet contains information necessary for routing, flow control, error checking, etc. in addition to the actual command and data. Since these overheads travel on the link along with the data and not on a dedicated command bus, they must be accounted for when choosing link parameters. That is, unlike a DDRx system which only transmits command and data, the HMC's packets contain overheads that reduce the effective raw bandwidth of the links. However, unlike a bi-directional bus, a full-duplex link does not require any "turnaround time" to change the direction of the bus. In a DDRx system with many, the utilization of the data bus can be significantly reduced by bus arbitration.

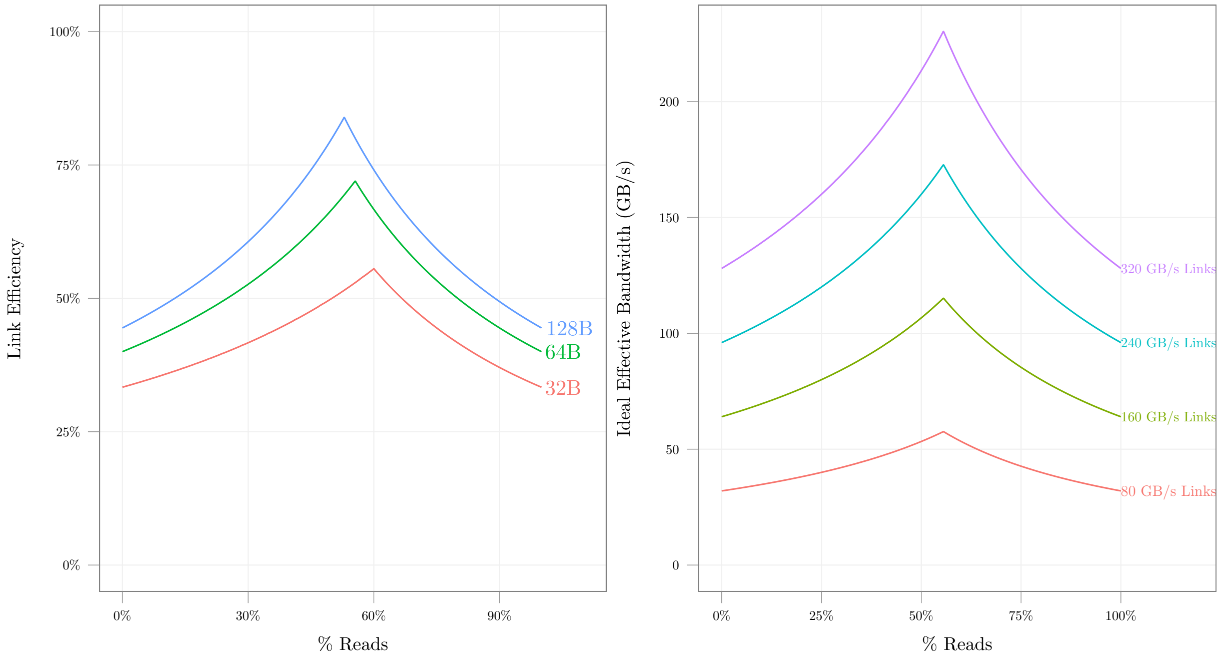
The bi-directional DDRx data bus is not sensitive to the read/write ratio of the stream since data flowing in either direction on the data bus represents useful bits. This is in contrast to the HMC where each link is composed of a dedicated request and response path that makes it sensitive to the read/write ratio: write data only flows on the request side of a link while read data only flows on the response side of a link. For example, if a memory access stream contains only writes, only the request side of the link will be utilized, while the response link will be idle[1]. There is an optimal read/write ratio at which both the request and response directions of a link are utilized fully, while other read/write ratios will leave the links partially idle.

We compute the theoretical peak link efficiency for different read/write ratios by considering how many overhead and idle cycles are introduced to transmit a stream with a particular read/write ratio. That is, we calculate the ratio of useful cycles on the link (i.e., cycles used to transmit data) to total cycles (i.e., all cycles including data, idle, and overhead cycles) as shown in equation 5.1.

$$Efficiency = \frac{Data\ Cycles}{Data\ Cycles + Idle\ Cycles + Overhead\ Cycles} * 100 \qquad (5.1)$$

It would be a reasonable expectation that the ideal read/write ratio for symmetric links should be 50%: write data would keep the request link busy while read data would keep the response link busy. Figure 5.1a shows the impact of the

[1] More precisely, the HMC will generate explicit write return packets when write returns cannot be piggy-backed on read returns, but these packets are pure overhead so we still consider the response link to be idle.



(a) Link efficiency as a function of read/write ratio

(b) Effective link bandwidth for 64 byte requests for different link speeds

Figure 5.1: Link efficiencies. Packet overhead is 16 bytes per packet. For a 64 byte request size, 56% reads yields a peak link efficiency of 72%.

read/write ratio on the peak link efficiency for varying packet sizes with a 16-byte packet overhead (as defined by the HMC standard). The interesting feature of this graph is that the peak efficiency for all request sizes is greater than 50% reads. This is due to the fact that a read request incurs a packet overhead twice (once for the request packet and once for the data response packet), and so the stream must contain a greater number of read requests to keep both sides of the link fully occupied. In other words, the response link only transmits read response overheads whereas the request link transmits write request overheads and read request overheads [2]. This means that the response link has more effective bandwidth to transmit read

[2] We assume that there is sufficient read traffic to piggy-back all write returns onto read response packets

Request Size	Peak Efficiency	Raw Bandwidth	Effective Peak Bandwidth
32B	55.6% at 60% Reads	80 GB/s 160 GB/s 240 GB/s 320 GB/s	44.4 GB/s 88.9 GB/s 133.3 GB/s 177.8 GB/s
64B	71.9% at 56% Reads	80 GB/s 160 GB/s 240 GB/s 320 GB/s	57.6 GB/s 115.1 GB/s 172.7 GB/s 230.2 GB/s
128B	83.9% at 53% Reads	80 GB/s 160 GB/s 240 GB/s 320 GB/s	67.1 GB/s 134.2 GB/s 201.3 GB/s 268.5 GB/s

Table 5.1: Effective link bandwidth for various request sizes and a 16 byte overhead. Larger requests achieve higher effective bandwidth on the links.

data.

Another salient feature of this graph is that as the request size becomes larger, the peak efficiency increases and shifts toward lower read/write ratios. The higher efficiency is due to the fact that the fixed 16 byte overhead is better amortized over the larger request size. The shift toward lower read/write ratios happens for the same reason: as a read overhead becomes smaller relative to read data, fewer reads are required to keep the response link fully utilized. Table 5.1 summarizes the effective peak bandwidth (corresponding to the peak of each curve in figure 5.1a) of various link configurations and packet sizes. As the table shows, packet overheads will always reduce the effective link bandwidth, making it impossible to achieve 100% link utilization. Thus the effective link bandwidth will always be lower than the raw link bandwidth. Link efficiencies of the read/write ratios used throughout this thesis are shown in table 5.2. In the next section we will use these observations to understand how to properly match TSV and link bandwidths for maximum utilization.

In order to achieve maximum link utilization, data should be flowing on both sides of a link at the same time. We have shown that link utilization is highly dependent on the read/write ratio, but we should note that the temporal structure of the memory access stream is also important to achieve full link utilization. That is, it is not enough for an overall memory stream to have a certain read/write ratio — the short-term read/write ratio should match the ideal read/write ratio at any given point in time to fully utilize the links. For example, a memory access stream that does 66 reads followed by 34 writes will only utilize one side of the link at a time and thus achieve much lower utilization than an access pattern that does 2 reads then 1 write repeatedly for 100 requests. Although both streams have the same overall read/write ratio and the same number of requests, the second stream will have much better link utilization.

5.2.2 Selecting Link/TSV Bandwidth

From the discussion in the previous section it can be concluded that in order to offset negative performance impact of link overheads, the link bandwidth should be greater than the aggregate bandwidth of the DRAM devices in order to maximize overall system-level throughput. We configure two typical HMC configurations which are discussed in [21]: 128 total banks in 16 vaults with 4 DRAM dies and 256 total banks in 16 vaults with 8 DRAM dies. Then we select an aggregate TSV

Read/Write Ratio	**Peak Efficiency**	**Raw Bandwidth**	**Effective Peak Bandwidth**
25%	50.0%	80 GB/s 160 GB/s 240 GB/s 320 GB/s	40 GB/s 80.0 GB/s 120.0 GB/s 160.0 GB/s
50%	66.7%	80 GB/s 160 GB/s 240 GB/s 320 GB/s	53.3 GB/s 160.7 GB/s 160.0 GB/s 213.3 GB/s
56%	71.4%	80 GB/s 160 GB/s 240 GB/s 320 GB/s	57.1 GB/s 114.3 GB/s 171.4 GB/s 228.6 GB/s
66%	60.0%	80 GB/s 160 GB/s 240 GB/s 320 GB/s	48.5 GB/s 97.0 GB/s 145.5 GB/s 193.9 GB/s
75%	53.3%	80 GB/s 160 GB/s 240 GB/s 320 GB/s	42.667 GB/s 85.3 GB/s 128.0 GB/s 170.7 GB/s

Table 5.2: Effective theoretical peak link bandwidths for different read/write ratios

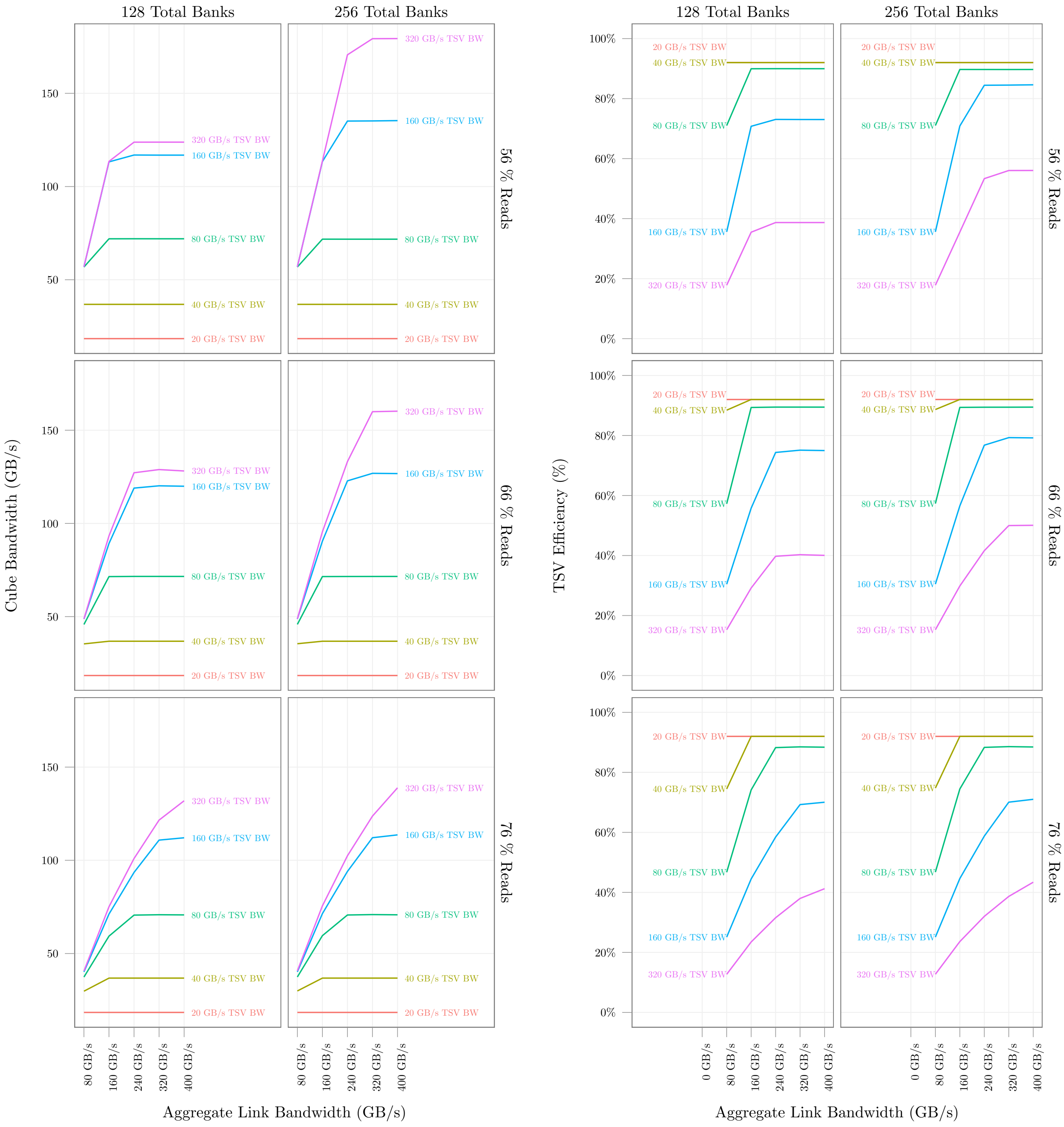
bandwidth and vary the aggregate link bandwidth based on typical values given in the HMC specification [22]. For this experiment, the switch parameters are set to provide effectively unlimited bandwidth between the links and the vaults so as to eliminate its impact. Figure 5.2a shows the results of several different link and TSV bandwidth combinations for three read/write ratios for the 128 and 256 total bank configurations.

As the link bandwidth increases, the overall throughput eventually flattens out indicating that the DRAM is unable to utilize the extra available bandwidth at the links. For example, considering the 56% reads case with 160 GB/s aggregate

TSV bandwidth, we see that the flat region begins at an aggregate link bandwidth of 240 GB/s. This is consistent with table 5.1: 240 GB/s links provide an effective bandwidth of 172GB/s, which is just higher than the 160 GB/s TSV bandwidth.

Since the link overheads are significant at 66% and 75% reads, higher link throughput is required to drive the DRAM to the peak bandwidth. This can be seen in the graph: as the read/write ratio increases, the flat area at the right of each graph moves farther to the right. However, if the TSV bandwidth is fixed at 160 GB/s, increasing the link bandwidth beyond 240 GB/s does not yield a significant advantage (except in the 75% case which must overcome a low effective link bandwidth). In all cases, increasing the TSV bandwidth beyond 160 GB/s yields diminishing returns (i.e., doubling the TSV bandwidth from 160 GB/s to 320 GB/s results in only a small performance increase). This is due to the fact that the memory banks are simply not fast enough to utilize the extra bandwidth. 256 banks are able to provide a higher degree of memory parallelism which are able to more effectively utilize higher TSV bandwidths.

Figure 5.2b displays the same data as Figure 5.2a except the main memory bandwidth is normalized to the aggregate TSV bandwidth. Normalizing the output bandwidth to the TSV bandwidth makes it much clearer which combinations of TSV and link bandwidths are the most efficient. Namely, if we only consider the raw output bandwidth, we neglect to take into account the cost of adding more TSVs or increasing the TSV clock rate. By normalizing the output bandwidth to the TSV bandwidth, it is easier to understand which configurations make the best trade-off between performance and efficient utilization of resources. This graph shows that



(a) Overall main memory bandwidth of several link and TSV throughputs.

(b) Main memory TSV efficiency ($TSV\ Efficiency = \frac{Main\ Memory\ Bandwidth}{Aggregate\ TSV\ bandwidth}$)

Figure 5.2: Several Link and TSV bandwidth combinations.

160 GB/s aggregate TSV bandwidth captures the optimal point as the maximum efficiency is nearly halved by increasing the TSV bandwidth further to 320 GB/s. The lowest link bandwidth that achieves maximum efficiency is 240 GB/s for 160 GB/s TSV bandwidth (i.e., the first point of the plateau on the 160 GB/s line is at 240 GB/s on the x-axis).

The other dimension in these graphs considers the impact of memory parallelism in the form of extra banks. Extra memory banks can help boost both the TSV utilization as well as overall performance (i.e., for a given line in the 128 bank panel, the corresponding line in the 256 bank panel to the right achieves both higher bandwidth and higher efficiency). Since, in this experiment, the extra memory banks are the result of adding extra DRAM dies to the stack (i.e., the 128 bank configuration has 4 stacked dies whereas the 256 bank configuration has 8 stacked dies), we can see that adding DRAM dies is an important capability in fully utilizing the available bandwidth of the HMC. However, even by doubling the number of banks, doubling the TSV bandwidth from 160 GB/s to 320 GB/s yields a only a 1.4x increase in overall performance with a drastic decrease in the TSV efficiency. This leads us to conclude that for a typical DRAM die configuration, a single cube can only efficiently utilize 160 GB/s of TSV bandwidth. This result matches figure 5.1a since the peak link bandwidth for 64 byte requests is 172.7 GB/s, which most closely matches the 160 GB/s TSV bandwidth. If we consider read/write ratios below 75%, most of the configurations can reach their peak efficiency with 240 GB/s of link bandwidth and 160 GB/s TSVs.

5.3 Switch Parameters

In the previous set of experiments, the switch settings provided effectively unlimited throughput between the high speed links and vault controllers. As mentioned before, the HMC specification does not describe a concrete switch architecture. However, it stipulates that a request from any link must be able to reach any vault or any other link. This condition simplifies the CPU interface, since the host can access the entire memory space from any link.

The crossbar switch architecture is simple and deterministic — it connects all links to all vaults and all other links such that a request from any link can proceed directly to any local vault (for a local request) or any other link (for a remote request in another cube). Since the amount of data within a given transaction is variable we allow for a single transaction to transmit for multiple cycles on a given switch path but transactions will not interleave (i.e., a transaction must finish completely before the data path can be switched to another destination or another transaction). For simplicity, we make the data path full-duplex: request packets flow on a different set of wires from response packets. This condition is necessary to avoid having to implement complex deadlock avoidance techniques inside of the switch. Finally, we assume that the switch path widths are uniform on both the transmitter and the receiver (e.g., if the switch transmits 2 flits out of the link buffer per cycle, the vault must also have a 2 flit-wide input data path).

Figure 5.3 shows how the switch data path width affects the output bandwidth. This result is to be expected, as the maximum bandwidth occurs when the switch

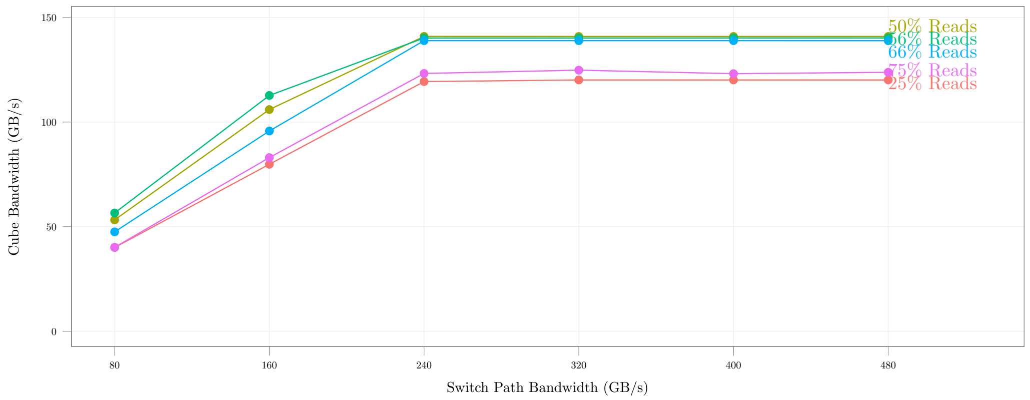


Figure 5.3: Main memory bandwidth with several different data path widths and read/write ratios. (64 entry Read Return Queue depth, 256 banks, 160 GB/s Aggregate TSV bandwidth, 240 Aggregate Link bandwidth)

bandwidth matches or exceeds the link bandwidth (i.e., each of the four links delivers 30 GB/s per direction and so each switch path should deliver 30 GB/s per second in each direction).

5.4 Queuing Parameters

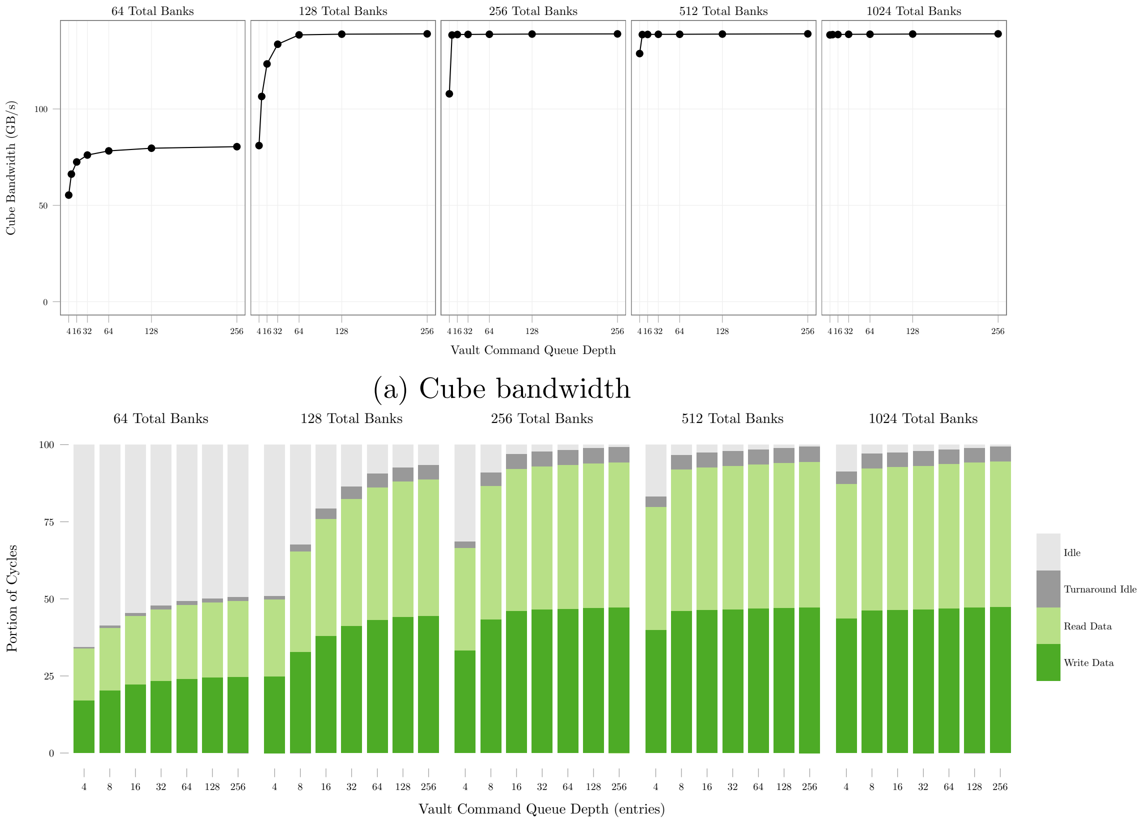
One aspect of the architecture that is worth exploring is the amount of queue space needed to optimize performance. In a memory system, queues are used for two primary purposes: collecting transactions that can be re-ordered in a more optimal way and holding data to even out bursty traffic patterns.

A typical memory controller will contain a command queue that can be used to schedule transactions around bank conflicts. That is, if two transactions are bound for the same bank, they must be serialized: the second must wait until the first completes before being scheduled for execution. If, however, a transaction is destined to the same vault but different bank, it can proceed independently of the stalled transaction (in a DDRx memory system, this same situation arises when a request is bound for the same rank but different bank). The situation can be even worse when the request is destined for a bank that is being refreshed, as refresh time can be quite long in a DDRx system. If a system has no buffering, a single stalled request can stall all accesses to the conflicting resource while others are under-utilized. In this case, the buffer is used to counteract hotspots caused by spatial locality.

In addition to command queueing, the second type of queue can be used to even out bursty traffic patterns. For example, if several read requests arrive at the same vault but are destined for different banks, they will all proceed largely in parallel (albeit, the controller still has to take into account the shared bus scheduling). Since the requests proceed in parallel, their data will become available at approximately the same time and should be buffered inside of a queue while it waits to return to the CPU. If no buffering existed, the controller would only be able to send a single read request in order to ensure no data is lost. In our model of the HMC architecture, this queue is called the "read return queue".

In the HMC example, there are many more vaults than high speed links. If, for example, we implement a round-robin switch arbiter with 16 vaults, a vault may have to wait for 15 other vaults to transmit data (in the worst case) before sending a second response. If the controller had to stall issuing all other read requests and wait for 15 vaults to transmit data, the memory system throughput would suffer significantly. In this case, the use of a buffer holds entries to help smooth out temporal hotspots.

Although queueing is helpful in these two scenarios, it is helpful to know the



(a) Cube bandwidth

(b) TSV Utilization

Figure 5.4: Effects of increasing command queue depth.

minimum amount of space needed to capture the most performance. Not only do queues take up die area, but deep queues could end up adding undesired latency. In this section, we will discuss the impact of queue sizes on HMC performance.

5.4.1 Vault Command Queue Depth

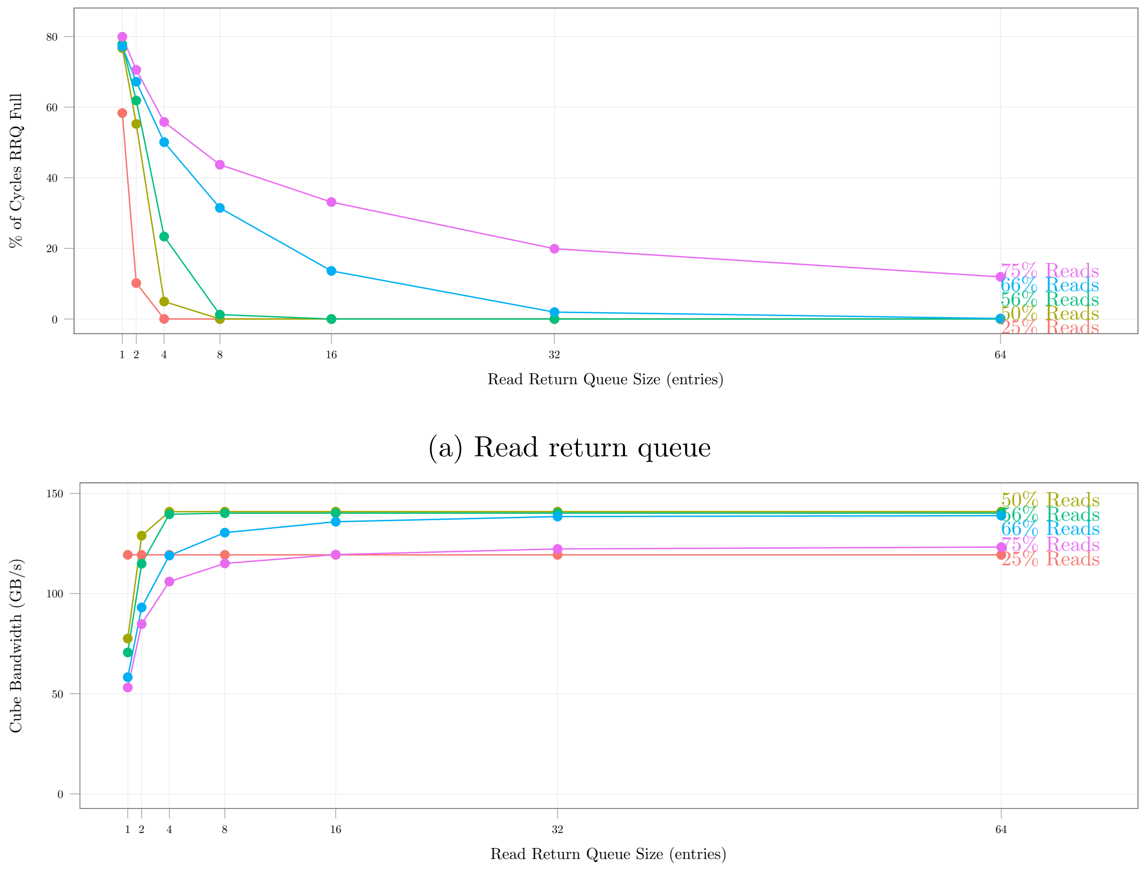
The vault command queue is used by the vault controller to reorder incoming DRAM transactions to schedule around DRAM conflicts. Figure 5.4a shows the impact of varying the command queue size fo a memory system with different numbers of total banks. While the 64 bank system is unable to reach the peak performance due to a lack of memory parallelism, all of the other configurations level off at the same maximum bandwidth for a given read/write ratio. One might expect that more banks per vault would require more queue space, but in fact, the relationship is the opposite. As the number of banks grows, fewer command queue entries are required to reach peak throughput. As described earlier, the primary goal of the command queue is to store entries for scheduling around bank conflicts. However, as the number of banks per vault increases, the likelihood of a bank conflict goes down, and so fewer queue entries are required to reorder requests.

It should also be noted that due to the nature of the “limit case” of the random stream simulations performed here, it is likely that a normal system would require fewer entries to capture the full performance of the vault. In other words, a 16 or 32 byte entry buffer should suffice for normal operation.

We can see this in 5.4b in that most of the idle time is due to bus arbitration (dark gray component) and not actual idle time (light gray component).

5.4.2 Vault Read Return Queue Depth

We repeat a similar experiment for the read return queue which stores return data inside of the vault as it awaits switch arbitration. Figure 5.5a shows the percentage of cycles the read return queue is full as a function of the RRQ depth. The data shows the behavior one would expect: for low read/write ratios, only a few entries are needed and the number of required entries goes up as more read requests are added to the stream. The rate of decay decreases with 75% reads and the RRQ is full some of the time no matter how many queue entries are added.



(a) Read return queue

(b) Cube bandwidth

Figure 5.5: Impact of increasing read return queue size.

Figure 5.5b shows the cube bandwidth for the same set of points. As expected, the output bandwidth is roughly inversely proportional to the portion of cycles that the RRQ is full. This is due to the fact that the controller must stall reads when the RRQ becomes full and so the DRAM throughput goes down.

5.5 Constrained Resource Sweep

To further understand how various design decisions impact a single cube's performance, we construct an experiment to evaluate a set of configurations that contain a constrained set of resources organized in different ways. That is, given a fixed aggregate TSV bandwidth, fixed available queue space in the logic layer, fixed switch bandwidth, fixed link bandwidth, fixed number of banks, etc., is there an optimal way to structure these resources? The main resource to hold constant is the number of banks as this defines the level of available parallelism in the memory system. Second, we hold the number of TSVs within the cube constant and divide them evenly among the vault. Since we keep the TSV data clock rate the same in all cases this results in a fixed aggregate bandwidth within the die stack. We also assume a fixed amount of buffer space such as the number of command queue entries and read return queue entries that will be shared among the vaults. Finally, we fix the aggregate switch bandwidth that the vaults must share.

5.5.1 Vault/Partition Organization

In this section, we quantify the trade-offs of taking a fixed set of banks and organizing them into different cube configurations. Using the discussion in section 5.2 as a starting point, we configure several different cubes that all have 240 GB/s aggregate link bandwidth and 160 GB/s aggregate TSV bandwidth. We vary the DRAM stack height, number of vaults, and banks per DRAM die to create configurations that have the same total number of banks, but arranged in different ways. For example, 128 total banks can be constructed with 16 vaults, 4 partitions (i.e., four stacked DRAM dies), and 2 banks per partition, or 128 total banks can be arranged as 4 vaults, 8 partitions, and 4 banks per partition, etc. Although these configurations have the same throughput and parallelism, their organization has an impact on the overall cube throughput.

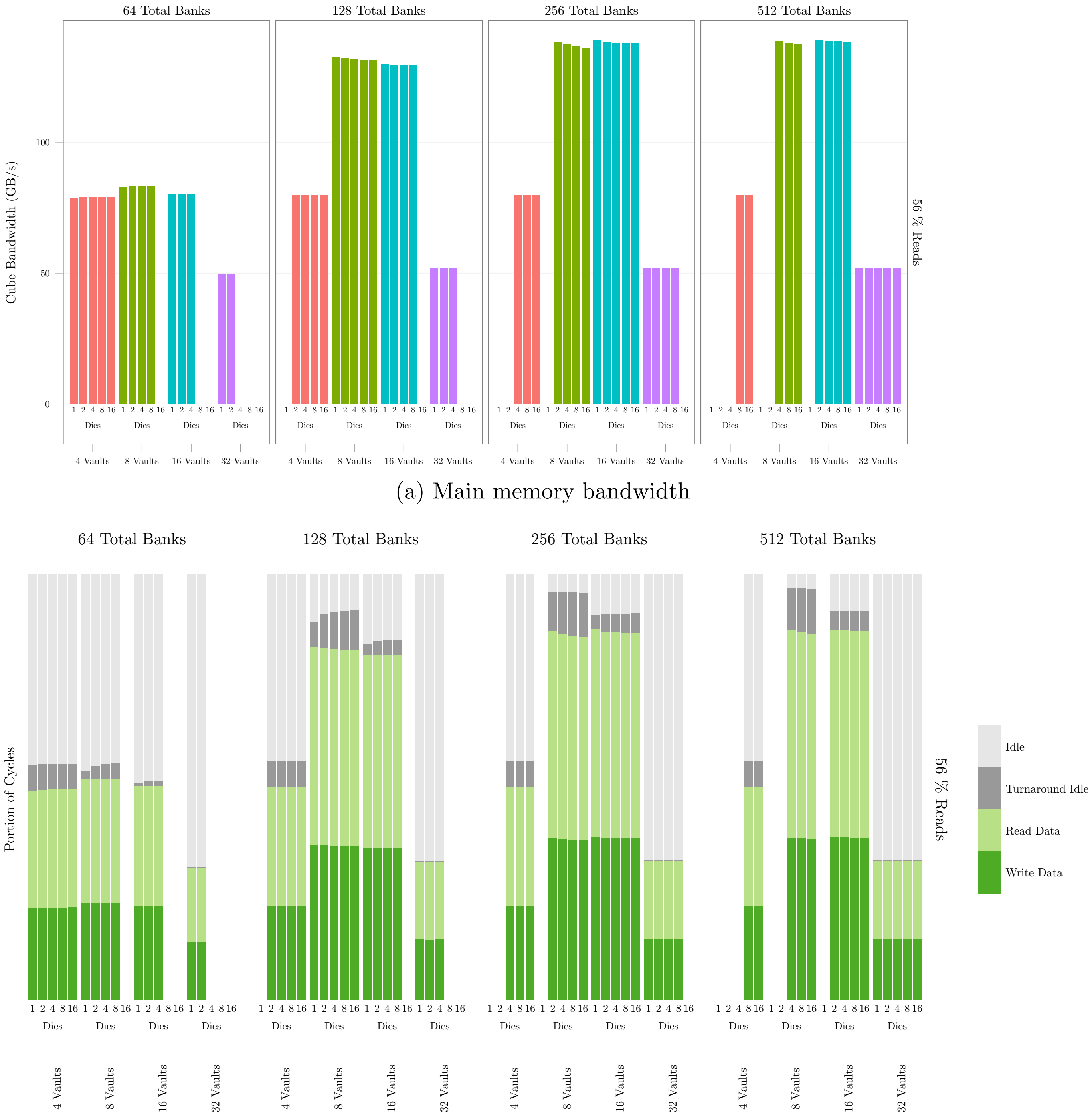
A configuration with a few vault controllers and many DRAM dies stacked on top would allow each vault controller to have extremely wide TSV buses to the memory banks, but such a configuration could potentially be limited by a lack of parallelism from having too few independent vaults (since vaults are roughly the equivalent of channels). At the other end of the spectrum, one can configure the same number of banks into many vaults and fewer dies stacked on top. This configuration would create more parallelism from independent channels at the expense of having a narrower data path between the controller and memory. In other words, we ask the question "for a given total number of banks, total number of TSVs, and fixed buffering and switch resources, is there a number of DRAM dies (and by extension, a number of vaults) that lead to the best performance?"

Note that in section 5.2.2 the switch bandwidth was effectively infinite so as to remove the impact of the switch. However, in this section, the aggregate switch bandwidth is finite and held constant among all configurations (i.e., doubling the number of vaults halves the switch bandwidth available for an individual vault). Additionally, we hold the total number of queue entries available in the logic layer to be constant (such as command queues and queues that hold return data) so that doubling the number of vaults halves the available queue space in each vault. By also holding the total number of banks constant and total number of TSVs constant, the bandwidth per bank is also identical for all configurations. Therefore, the performance differences are solely the result of the interaction between the various components.

First, we drive the cube configurations with a 56% read/write ratio stream since this stream results in the highest effective link bandwidth. Figure 5.6a shows the results for cubes consisting of 64, 128, 256, and 512 total banks. Bars are grouped by the number of vaults, and each bar within a group represents a different number of DRAM dies in the stack. Configurations requiring fewer than one bank per partition or greater than 16 banks per partition are considered invalid and so each group may have missing bars.

The most noticeable feature of this graph is that within each group of bars, the output bandwidth of the memory system is similar. This points to the fact that the most important determinant of the memory system bandwidth is the number of vaults (for a given number of total banks). In this experiment the number of vaults determines both the queue space per vault controller as well as the bandwidths of each TSV bus attached to it (i.e., with 160 GB/s of aggregate TSV bandwidth available per cube, 8 vaults would have access to 20 GB/s TSV bandwidth per vault and 16 vaults would have 10 GB/s per vault of bandwidth to the memory devices). Both 8 and 16 vaults are able to achieve the highest output bandwidth for all of the configurations showed.

Within each group of bars the performance is similar (i.e., for a given number of vaults, distributing the banks among different numbers of stacked dies yields similar performance). This indicates that the performance overhead of die stacking is in fact quite low. This experiment shows that stacking a larger number of lower density dies can achieve similar performance and capacity as stacking a smaller



(a) Main memory bandwidth

(b) TSV Utilization

Figure 5.6: Performance of several resource constrained cube configurations organized into different numbers of vaults, partitions, and total banks.

number of higher density dies. In other words, if the reliability and yields of die stacks are improved to allow building taller stacks, the industry can overcome the predicted end of DRAM scaling by simply adding more dies to the stack. In doing so, one can increase the overall capacity and memory-level parallelism of the device while keeping the density of an individual DRAM die constant. Moreover, it stands to reason that the TSV stacking processes will only improve as vendors move to mass produce stacked parts and decrease their cost.

To obtain a more detailed picture of the internal dynamics of the cube, we track the utilization of each TSV bus by counting the number of cycles a TSV is transmitting data or being idle. Figure 5.6b shows the same configurations as 5.6a where the average TSV utilization components are averaged among the different vaults. The green components represent the portion of cycles spent transmitting useful data (reads and writes) while the gray components represent two kinds of idle times when the TSVs are not being used to send data. The "turnaround idle" component represents the equivalent of rank-to-rank switching time and write-to-read turnaround time in a traditional DDRx system. Since partitions are analogous to ranks in a traditional DDRx system (i.e., multiple partitions share a common TSV bus), the simulator adds an idle cycle between back-to-back data bursts from different partitions and between consecutive reads and writes. The final idle component represents the cycles where, for whatever reason, the TSVs are not being used to transmit any data. Such idle cycles may arise due to bank conflicts, insufficient request rate to the vaults, etc.

As the number of DRAM dies in the stack grows, there is a small increase in the turnaround idle component indicating that more back-to-back reads to different partitions and read-to-write requests are issued. This is to be expected as the probability of reads to different partitions grows with the number of partitions and thus requires more turnaround cycles to be inserted. This accounts for the small bandwidth decrease within a particular group of bars: more TSV bus arbitration between a larger number of dies connected to that bus.

As the number of vaults decreases from 16 to 8, the turnaround component (dark gray portion) increases significantly. Compared to a 16 vault configuration, each vault in the 8 vault configuration must service twice as many requests with twice as much available TSV bandwidth. The increased request load leads to a higher probability of needing to insert turnaround cycles. The problem is further accentuated by the fact that the relative cost of idling a wider bus (when fewer vaults are present) for a single turnaround cycle is higher than idling a narrower bus for a single cycle (i.e., if a request typically takes n cycles followed by a cycle of turnaround, doubling the bus throughput will reduce the data time to $n/2$ cycles while keeping a single cycle turnaround penalty). These two factors together account for the increase in the relative number of turnaround cycles when going from 16 to 8 vaults.

The four vault configuration does not perform well due to the fact that it has very few wide TSV buses. As mentioned previously, the cost of an idle cycle is high with a wide bus since the number of bits that could have been transmitted that cycle is high. In other words, every idle TSV cycle in the four vault configuration has twice the impact on the TSV efficiency as in the eight vault configuration. The overall result is that, although the vault controllers in the four vault configuration can receive requests quickly (high per-vault switch bandwidth) and transmit them quickly to the memory devices (high TSV bandwidth), the cost of idling the bus is simply too high to maintain a reasonable efficiency.

The wide TSV buses are also inefficient because in essence, they transfer data to and from the DRAM too quickly. Although this seems slightly counter-intuitive, this is due to the fact that the DRAM timing constraints stay the same in our case while the data burst time decreases significantly. This results in a situation where the I/O is not effectively overlapped with the DRAM access time in addition to the high turnaround penalty.

Within the 32 vault configuration, there are two main factors that cause poor performance: switch bandwidth and queue depths. Since we constrain the aggregate switch bandwidth and the total available queue depth in the logic layer, the 32 vault configuration has limited per-vault switch bandwidth as well as limited per-vault queue space for incoming requests and outgoing return data. This creates a situation where each vault controller does not have a large enough pool of requests to schedule from as there are limited queue slots that are slowly filled by the limited switch bandwidths. Similarly, return data streaming from the vaults drains slowly; if too many return requests build up in the vault controller, the simulator stalls issuing new requests to prevent data loss. In this way, the limited queue space can adversely impact the flow of memory requests through the memory banks.

5.5.2 Impact of Total Banks

Another aspect of the constrained resource sweep is the impact of the total number of banks on overall performance. As discussed in the previous section, the 4 and 32 vault configurations are unable to make efficient use of the DRAM and so are not sensitive to the total number of banks. Specifically, when we move from 64 to 512 total banks in figure 5.6a, the 4 and 32 vault configurations show almost no increase in overall bandwidth. If adding memory-level parallelism to the cube does not increase the overall bandwidth, this is an indication that the bottleneck occurs somewhere outside of the DRAM devices (i.e., in the switch or scheduling processes).

The middle two graphs show that the 8 and 16 vault configurations can achieve almost peak performance using 128 banks. In fact, increasing the total number of banks to 256 yields a 4.2% and 6.5% increase in throughput in the 8 and 16 vault cases, respectively. Since each bank requires extra circuitry (sense amplifiers, row decoders), a trade-off could be made to build 128 larger banks with less overall circuitry instead of 256 smaller banks. This 128 bank system would provide enough parallelism to capture a large portion of the available throughput while reducing DRAM die complexity and power. Furthermore, some applications may be unable to utilize more than 100 GB/s of memory bandwidth and thus one could reduce the bank-level parallelism and the aggregate throughput to save on cost and power budgets.

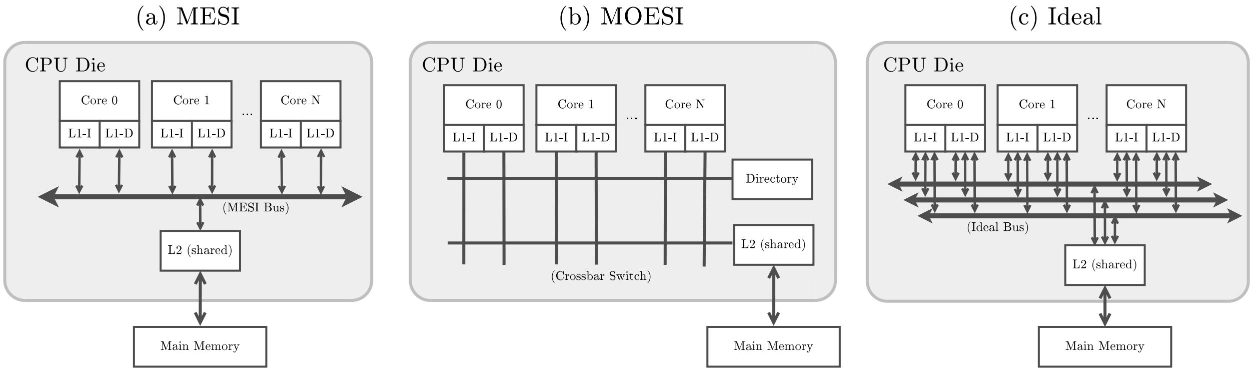
In conclusion, this section has highlighted some important issues in the design space of the HMC. The effective link bandwidth is sensitive to the read/write ratio and packetization overhead which requires links to have significantly higher throughput than the TSVs. Adding more capacity to the cube can be achieved by stacking more DRAM dies with only a small performance penalty due to increased turnaround overhead. With 160 GB/s TSVs, a 16 vault configuration is able to reduce the impact of turnaround idle time and make the most effective usage of the TSVs. Finally, in order to utilize 160 GB/s TSV bandwidth, at least 128 banks are required but having 512 banks does not improve performance further.

5.6 Full System Simulation

5.6.1 Memory Bandwidth Exploration

Although many papers lament the lack of memory bandwidth as a system-level bottleneck in modern systems, testing this assertion in a simulation setting turns out to be somewhat difficult. Conventional wisdom states that modern multi-core architectures demand more memory bandwidth since each additional core adds parallel computation that in turn adds traffic to the memory system. If the main memory is not able to provide the extra bandwidth required, the overall performance of the system will suffer. As the random stream simulations in the previous sections have shown, the HMC architecture can potentially provide an order of magnitude more bandwidth than a typical DDRx memory system (i.e., over one hundred versus dozens of gigabytes per second). However, our initial full system simulations showed that adding main memory bandwidth did not result in a significant reduction in execution time. Closer inspection of the statistics from the memory simulator confirmed that the CPU simulator was not able to generate nearly enough bandwidth to stress the HMC for a variety of workloads.

To make matters worse, adding extra cores and running multi-threaded workloads that utilized all of the available cores failed to substantially increase the main memory bandwidth demand as well. After careful examination of the bindings between the CPU and memory simulators, we discovered that the source of the problem stems from the cache coherence protocol on the CPU: as the number of cores and number of memory operations scaled up, the default MESI coherence scheme of the



(a) MESI

(b) MOESI

(c) Ideal

Figure 5.7: Comparison of coherence schemes. In the “ideal” scheme, the bandwidth between the private L1s and shared L2 scales proportionally to the number of cores.

MARSSx86 simulator was creating a bottleneck. The coherence traffic on the bus overshadowed actual useful traffic and the throughput of requests out of the CPU was greatly diminished. This was a big problem for our simulation infrastructure: if the CPU cannot take advantage of the available memory bandwidth due to bottlenecks on the CPU die, there would be no way of showing the benefits of a new high bandwidth memory technology such as the HMC.

In order to quantify the extent of the problem, we simulate the execution of the memory-intensive STREAM [49] benchmark in MARSSx86 with the “perfect” memory model described in section 4.4.3. We vary the core count (2, 4, 8) and cache coherence schemes (MESI, MOESI, and Ideal). The standard MESI bus structure uses a round-robin algorithm to transmit a single request every two cycles. The MOESI structure connects the caches to a directory using a crossbar switch. The ideal bus can transmit one request per core every 2 cycles. Since the bandwidth of the ideal structure is proportional to the number of cores, it removes the bottleneck of the MESI structure. A comparison of the coherence structures is shown in figure 5.7. It is important to note that the Ideal bus structure, while based on the MESI

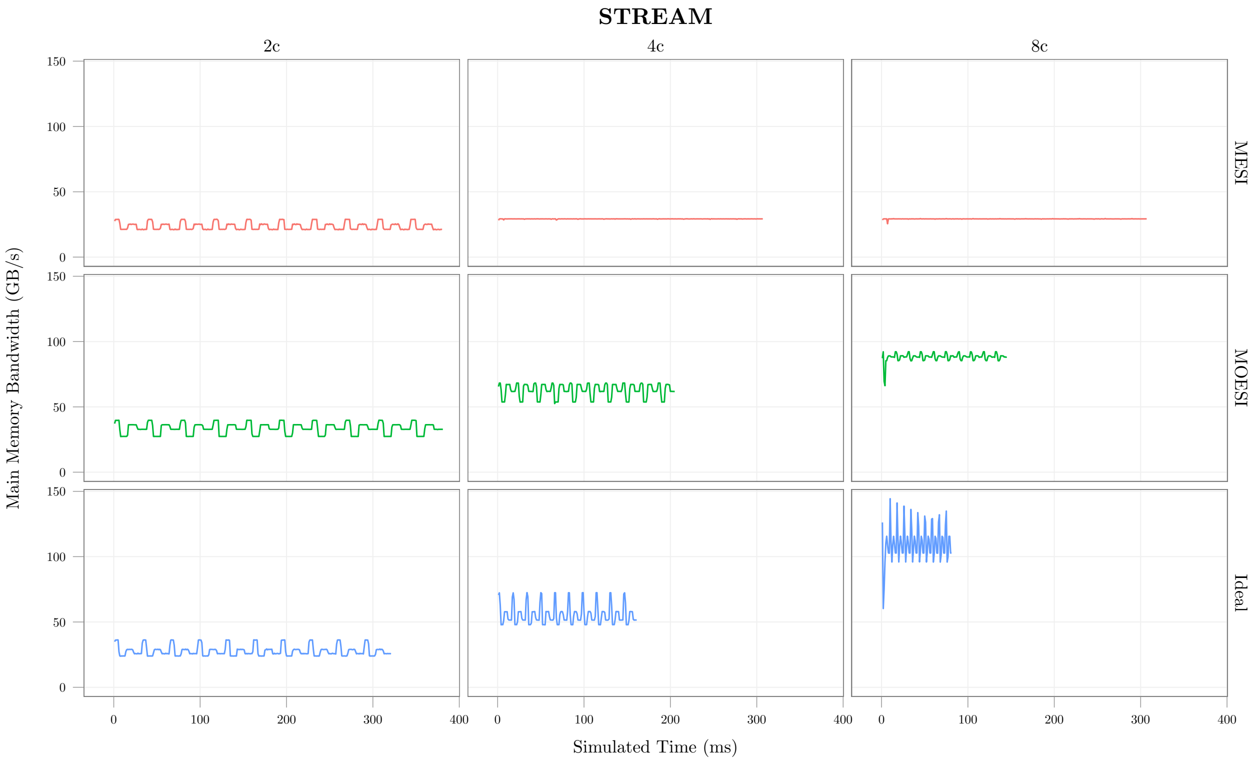


Figure 5.8: Core scaling of the STREAM benchmark with various coherence schemes. The MESI coherence scheme restricts core scaling.

bus structure, is not coherent.

Figure 5.8 shows the time-series execution of the STREAM benchmark for the various combinations of cores and coherence schemes. As all of these executions are between the same two region of interest points in the program, a shorter line indicates a lower execution time. The first feature that is immediately apparent from the graph is the lack of memory bandwidth scaling in the MESI bus: doubling the number of cores results in almost no increase in the off-chip memory bandwidth. This is because the bottleneck occurs within the MESI protocol long before a request reaches the main memory system. It also becomes evident that this memory bottleneck limits the usefulness of additional cores as they simply spend their cycles waiting to access the MESI bus. Another interesting feature of the MESI graph is that the periodic features of the STREAM benchmark flatten out when scaling beyond two cores. This indicates that with only two cores, some phases of the benchmark have a lower bandwidth demand than what the MESI bus provides; however, as more cores are added, every phase of the benchmark exceeds the effective bandwidth of the MESI bus and the line becomes flat.

The MOESI coherence scheme scales better than MESI, but sub-linearly with the number of cores (i.e., going from two to eight cores yields only a 1.58x speedup). Given the structure of the ideal bus, it scales roughly linearly from 2 to 4 cores, but fails to scale beyond 4 cores. This is due to the fact that although the bus bandwidth scales with the core count, it still has the limitation that it can only transmit when the memory controller pending queue has room. See table 5.3 for the full list of speedups over the dual core system.

The standard STREAM workload has an access pattern that sequentially accesses large arrays of `double`s that result in much larger working sets than any standard on-chip cache (for example, our configuration accesses approximately 180 MB of memory). Although this access pattern results in many capacity misses, a 64 byte cache line still yields 7 hits for each capacity miss. Since our goal is to maximally stress the main memory system, we modify the STREAM benchmark to eliminate these cache hits by padding each element in the array to the width of a full cache line. Each access to the arrays only touches the first `double` (8 bytes) of the cache line and the rest of the line is not used. This creates an access stream where all of the memory accesses are a cache miss. We refer to this benchmark as the “main memory” STREAM or STREAM-mm. A comparison of the sum phase of STREAM and STREAM-mm is shown in figure 5.9.

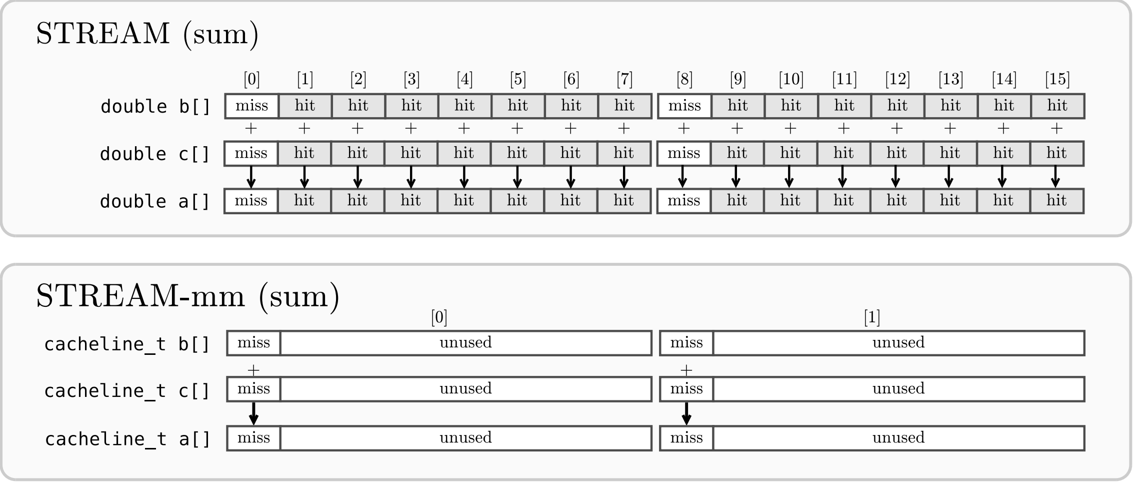


Figure 5.9: Comparison of the access patterns of STREAM and STREAM-mm. By padding each element, STREAM-mm purposely removes all of the cache hits from STREAM's memory access stream and maximizes stress on the main memory system.

We repeat the coherence scaling experiment as before but this time we use the STREAM-mm workload. It is clear that the MESI bus structure shows no memory bandwidth increase or execution time decrease as a result of adding cores. Since STREAM-mm is designed to increase the main memory pressure by removing the role of the cache, it would be reasonable to assume that the main memory bandwidth should increase given that the ideal memory model can supply infinite bandwidth. To the contrary, we see that the STREAM-mm memory bandwidth is lower with the MESI and MOESI coherence schemes as compared to STREAM. We conclude, then, that the overhead of the coherence protocol actually diminishes the off-chip bandwidth demand. The "Ideal" structure, on the other hand, removes the coherence bottleneck and is able to generate hundreds of gigabytes per second of request bandwidth to the perfect memory model.

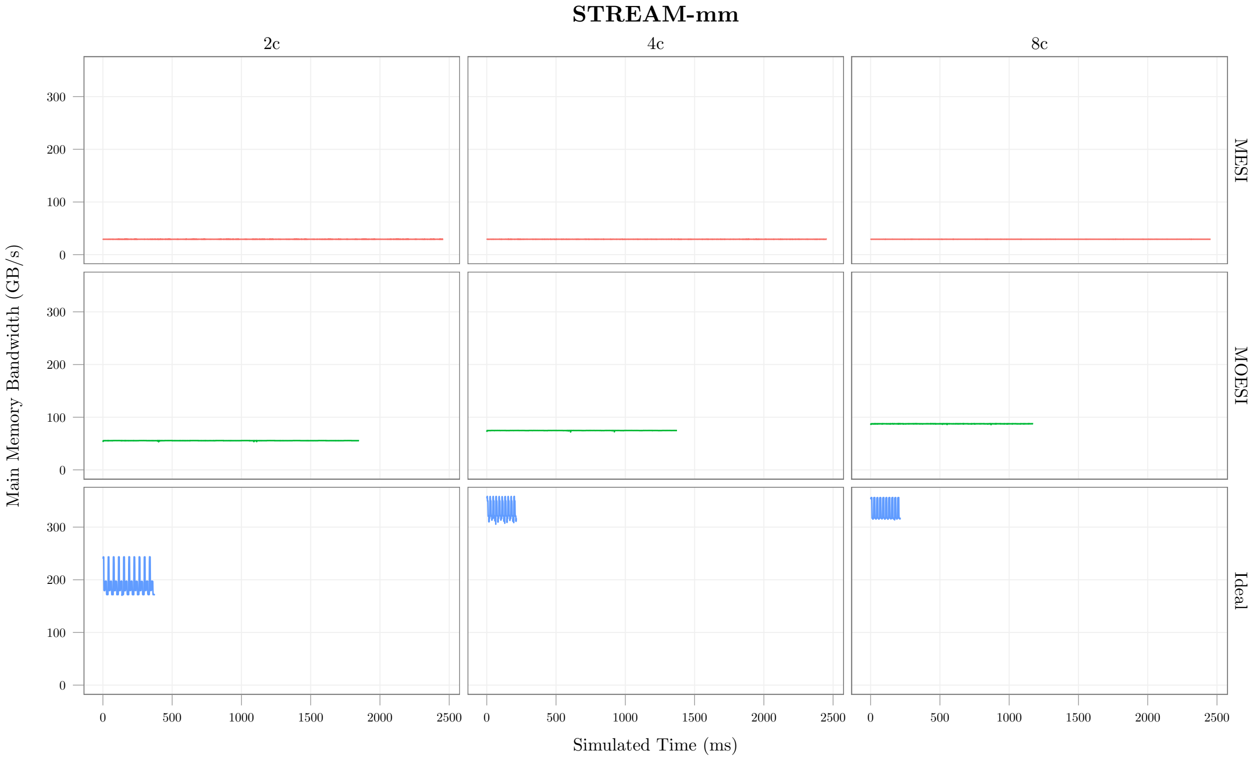


Figure 5.10: Core scaling of the STREAM-mm benchmark with various coherence schemes. The main memory bandwidth of the MESI and MOESI protocols actually decreases when running STREAM-mm instead of STREAM. The Ideal bus removes the coherence bottleneck and floods the memory system with requests.

	STREAM			STREAM-mm		
Cores	Ideal	MESI	MOESI	Ideal	MESI	MOESI
2	1.00	1.00	1.00	1.00	1.00	1.00
4	1.99	1.24	1.86	1.72	1.00	1.35
8	3.96	1.24	2.63	1.72	1.00	1.58

Table 5.3: Speedup of workloads when increasing core count and changing coherence scheme. Speedups are computed based on the dual core execution time.

Since the HMC can potentially deliver an order of magnitude more bandwidth than previous memory systems, its benefits can only be demonstrated with workloads and CPU structures that can take advantage of the available memory bandwidth. The previous scaling experiments have shown that it would be difficult to scale the main memory bandwidth high enough to saturate the HMC using the MESI or MOESI protocols. Therefore, we choose the ideal coherence bus model in all of the full system experiments in this dissertation. In other words, to show the full benefit of the HMC on system performance we need to move the memory bottleneck past the cache and back to the main memory.

These experiments also demonstrate that the industry may be reaching a point where on-chip cache structures should be redesigned to support next generation memory systems. Specifically, in order to embrace next generation memory architectures, it may be necessary to depart from traditional coherence guarantees and move to weaker consistency or non-coherent memory models that will allow the cores to take advantage of available memory bandwidth. However, such a move will be difficult as writing parallel programs has a steep learning curve even with the strong coherence guarantees of today's CPUs.

5.6.2 Workload Selection

As the previous section showed, it is not always trivial to find a set of workloads that will generate enough off-chip bandwidth to utilize the HMC's available bandwidth. This is problematic when attempting to characterize the impact of design choices in full system simulation: if the workload cannot utilize the memory system, it is likely that memory systems design choices will not meaningfully affect the overall execution time of the program. In order to avoid simulating workloads that are unable to utilize any significant amount of memory bandwidth, we attempt to characterize the memory behavior of several multi-core workloads from various sources: multi-threaded benchmarks from the PARSEC [50] and NAS Parallel Benchmark [51] suites, several micro benchmarks such as GUPS and STREAM [49] (and our STREAM-mm variant described earlier), and one MANTEVO mini application [52]. Since full-system simulation with the CPU and memory system forms a closed feedback loop, sometimes it is difficult to separate the effects of the memory system on the simulation. As in the previous section, we use our previously described "ideal" memory model which provides infinite bandwidth to capture the off-chip bandwidth of several applications running using eight threads on an eight core CPU model. To reduce simulation time, the workloads are run for a maximum of 2 billion cycles (or until completion).

Figures 5.11, 5.12, 5.13, and 5.14 show the time-series bandwidths of the execution of the workloads from the PARSEC suite, NAS Parallel Benchmark suite, the synthetic micro benchmarks, and MANTEVO mini apps, respectively. The interesting thing depicted among these time-series graphs is the variety of access patterns that are exhibited by these applications. Some applications are highly periodic (such as STREAM, fluidanimate, ua.C, miniFE), while others make steady use of the main memory (ex: dc.A, eg.C, canneal, portions of streamcluster), and others have very irregular access patterns that look very noisy (dedup, GUPS, ep.C). Although the

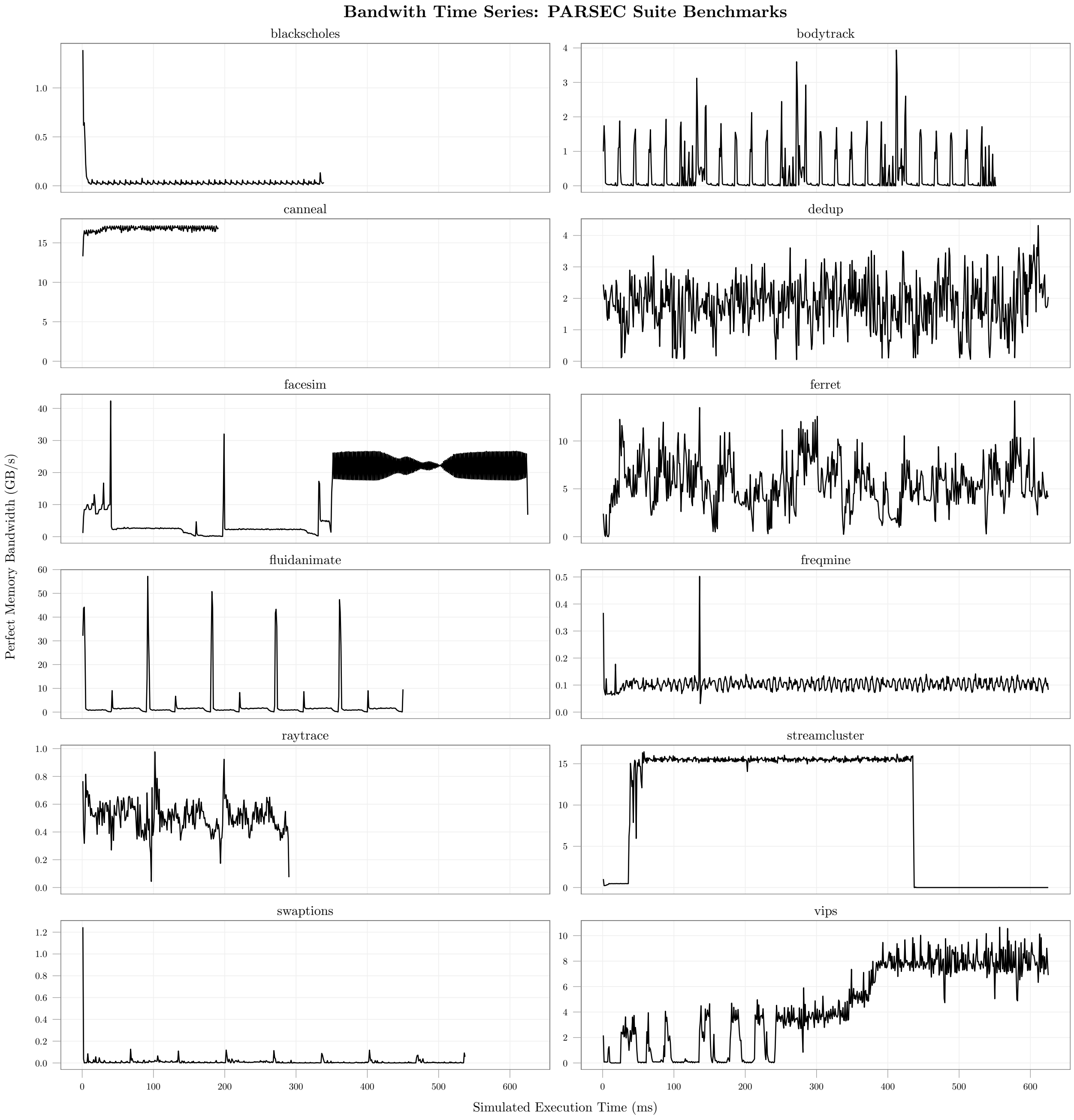


Figure 5.11: Bandwidth time series: PARSEC suite.

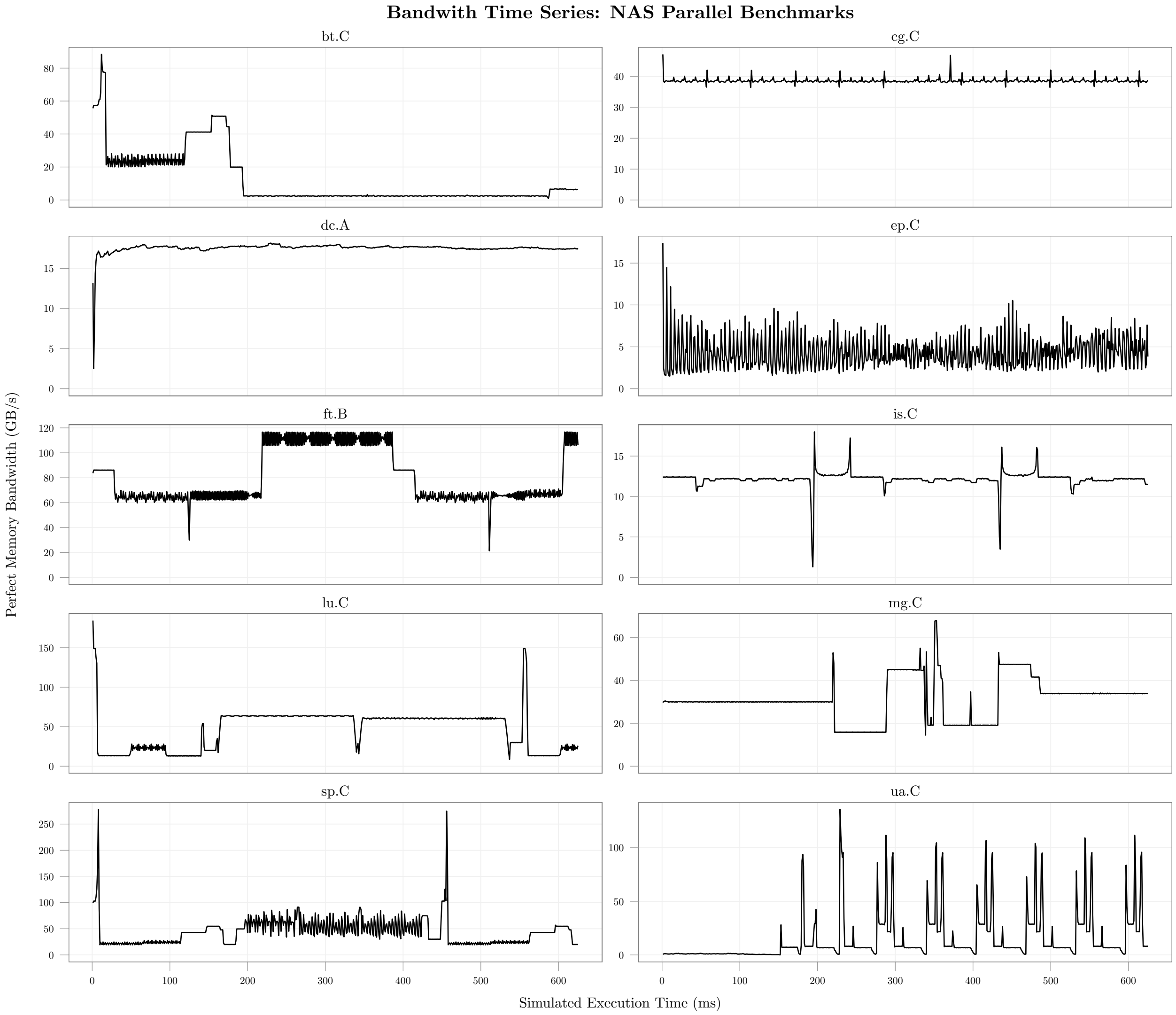


Figure 5.12: Bandwidth time series: NAS Parallel Benchamark suite.

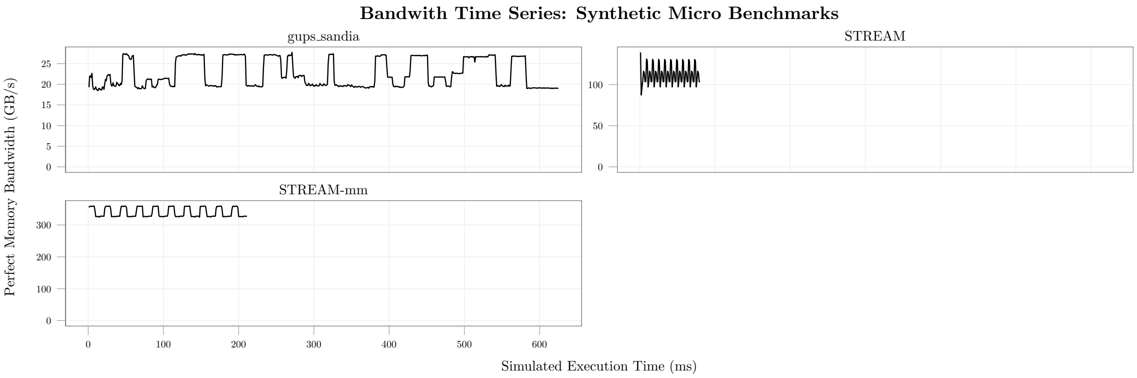


Figure 5.13: Bandwidth time series: synthetic micro benchmarks

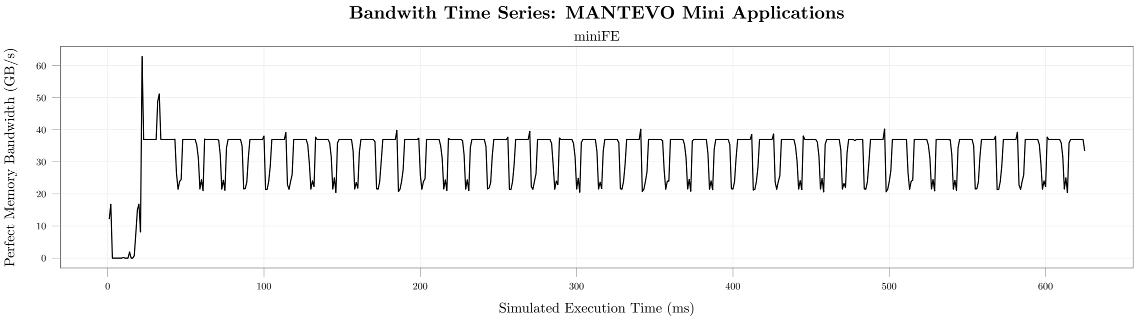


Figure 5.14: Bandwidth time series: MANTEVO mini application (MiniFE).

access patterns are very different among the workloads, it becomes clear that many workloads do not generate any significant traffic to main memory and so would not be useful to simulate in full-system with the HMC model.

To summarize the access patterns of these workloads, we construct a boxplot that shows the bandwidth distribution of each of these workloads shown in figure 5.15 The lines of each box extend to the minimum and maximum bandwidths for that workload and the bandwidths are sorted by their average bandwidth. STREAM and STREAM-mm are, of course, the workloads that generate the highest bandwidth to memory. Fluidanimate, ua.C, and sp.C have low average bandwidth requirements but with periods of high bandwidth bursts.

5.7 Full System Results

5.7.1 DRAM Sensitivity

As mentioned previously, past generations of DRAM have been simple “dumb” devices that contain almost no logic and share a common data bus. In these devices, the performance of the main memory system as a whole is directly proportional to

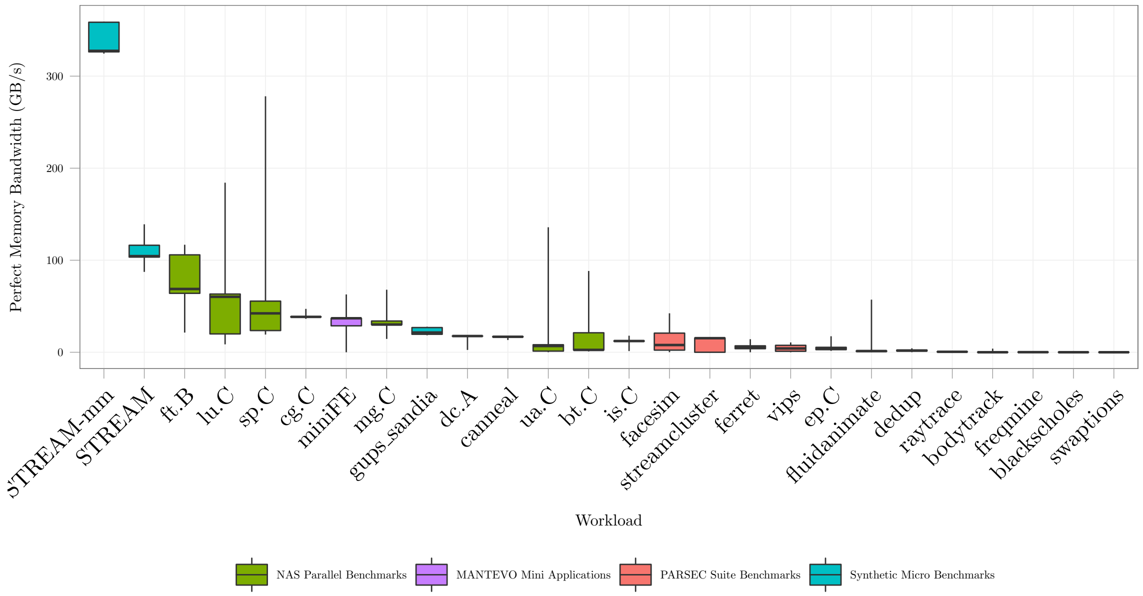


Figure 5.15: Box plot summary of the bandwidth characteristics of all workloads. Workloads are sorted in decreasing average bandwidth and color coded by their respective benchmark suites.

the performance of the memory devices connected to the data bus. However, the HMC introduces extra parallelism which makes the performance of any individual device less critical. That is, many memory banks and many memory controllers decrease the likelihood of bank conflicts and allow many requests to be in-flight at any given time. Having many in progress requests takes any individual device off the critical path and allows high throughput to be maintained even when memory devices are slower.

In order to test this hypothesis, we configure several identical HMC configurations with varying DRAM timings. We modify two key parameters that affect the performance of the DRAM: t_{RAS} and t_{RCD}. For the base case, we use the timings described in 4.2.1: 34 cycles and 17 cycles for t_{RAS} and t_{RCD}, respectively. Each value is then halved and doubled and the cross product of the timings are tested.

The values are constrained to only include the cases where $t_{RAS} > t_{RCD}$. This is due to the fact that the t_{RCD} constraint can extend the row cycle time beyond the usual value of $t_{RAS} + t_{RP}$. If t_{RCD} is greater than t_{RAS}, any changes to t_{RAS} would not change the timing of the end of the DRAM cycle. So instead of the full cross product of 27 simulations, we only consider the results for the 18 valid combinations of these parameters.

In closed page mode, both of these timing parameters have an impact on the maximum number of requests per unit time that can be serviced by a given bank. The t_{RAS} parameter represents the amount of time that it takes for a row access to restore data to the DRAM array [5]. A bank cannot be precharged until t_{RAS} after a row activation. The next request cannot proceed until the row has been precharged (in closed page mode). As can be seen in figure 5.16, increasing t_{RAS} has the effect of pushing the precharge forward in time and thus delaying the beginning of the next request (ACT B).

In addition to modifying t_{RAS}, we also modify the t_{RCD} parameter. t_{RCD} represents the amount of time necessary to perform the sensing of the data from the DRAM array. That is, a column access command to read or write data cannot proceed until t_{RCD} time after a row activation.

Similarly to increasing t_{RAS}, increasing t_{RCD} limits the available bank bandwidth because it delays the implicit precharge command that follows the READ command in closed page mode. This has the effect of delaying the start of the next request (ACT B), increasing the time to service a request, and thus lowering the bank bandwidth. Moreover, as shown in figure 5.17, increasing t_{RCD} also has the

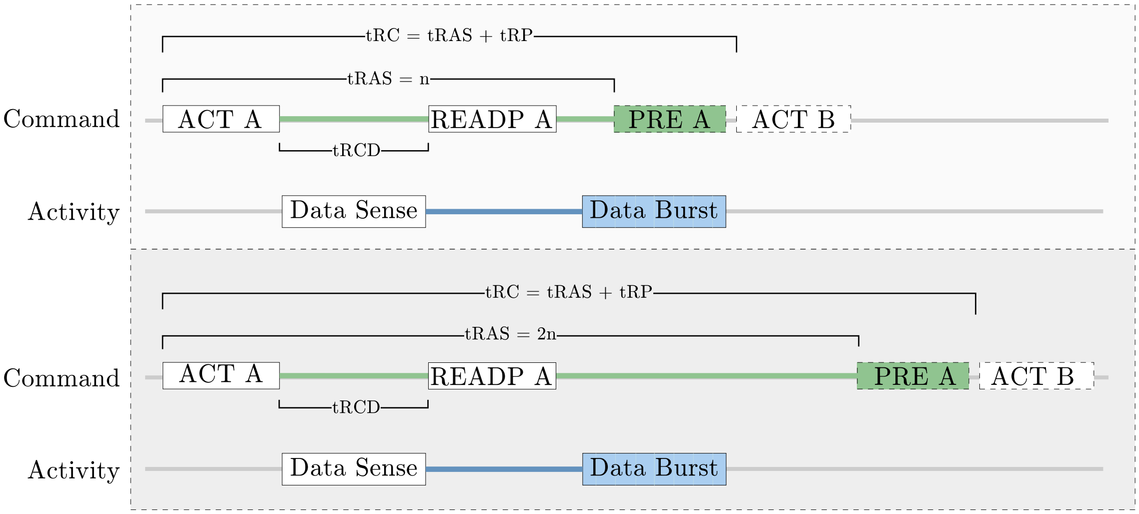


Figure 5.16: Effects of doubling the t_{RAS} DRAM timing paramter. Although the data burst comes out at the same time in both cases, the start of the next request is delayed when t_{RAS} is doubled.

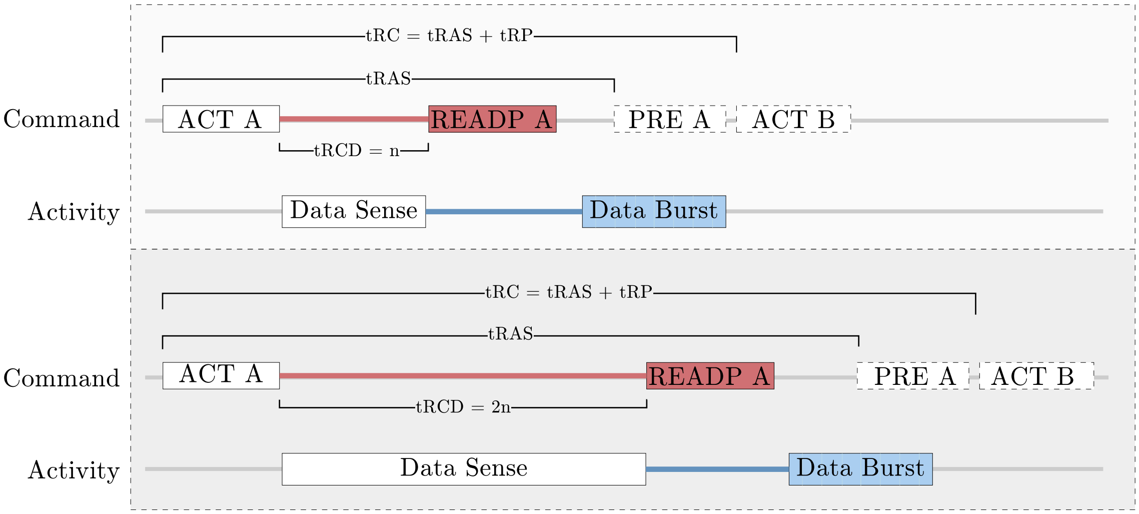


Figure 5.17: Effects of doubling the t_{RCD} DRAM timing paramter. By delaying the start of the column access, the bank bandwidth is decreased. Furthermore, since the beginning of the data burst is delayed, the latency of requests is increased.

effect of delaying the data burst out of the bank. This means that increasing t_{RCD} has an added penalty in that it increases the latency of every memory request in the system. For latency-sensitive applications, the extra time spent waiting for the data could cause a significant perturbation in system performance.

First, we plot the bandwidth time series for three workloads for the valid combinations of parameters for a 128 bank HMC in figure 5.18. At first glance, it appears that although the DRAM timing parameters are changing drastically, the overall performance does not suffer significantly. Then, as a comparison point, we also graph the bandwidth over time for the same DRAM configurations, but with a 256 bank HMC in figure 5.19. As we have seen previously, the 256 bank configuration is able to achieve marginally better overall bandwidth for the most memory intensive workload (STREAM-mm). To get a better summary of the results, we condense the time series into a set of box plots shown in figure 5.20.

These results are fairly surprising at first glance: two crucial DRAM timing parameters change by a factor of two and yet it does not appear to have a very large impact on the overall output bandwidth. In order to check these results, we also plot the average latency components of requests as they move through the cube during execution of the STREAM benchmark in figure 5.21.

Indeed the pattern is consistent with the expected behavior shown in figures 5.16 and 5.17. The non-DRAM components of the latency are stable among the different configurations while the DRAM latencies vary. The blue line corresponds to the latency component between the time when a request is sent on the TSVs and when it is received in the read return queue (i.e., the actual DRAM access and data

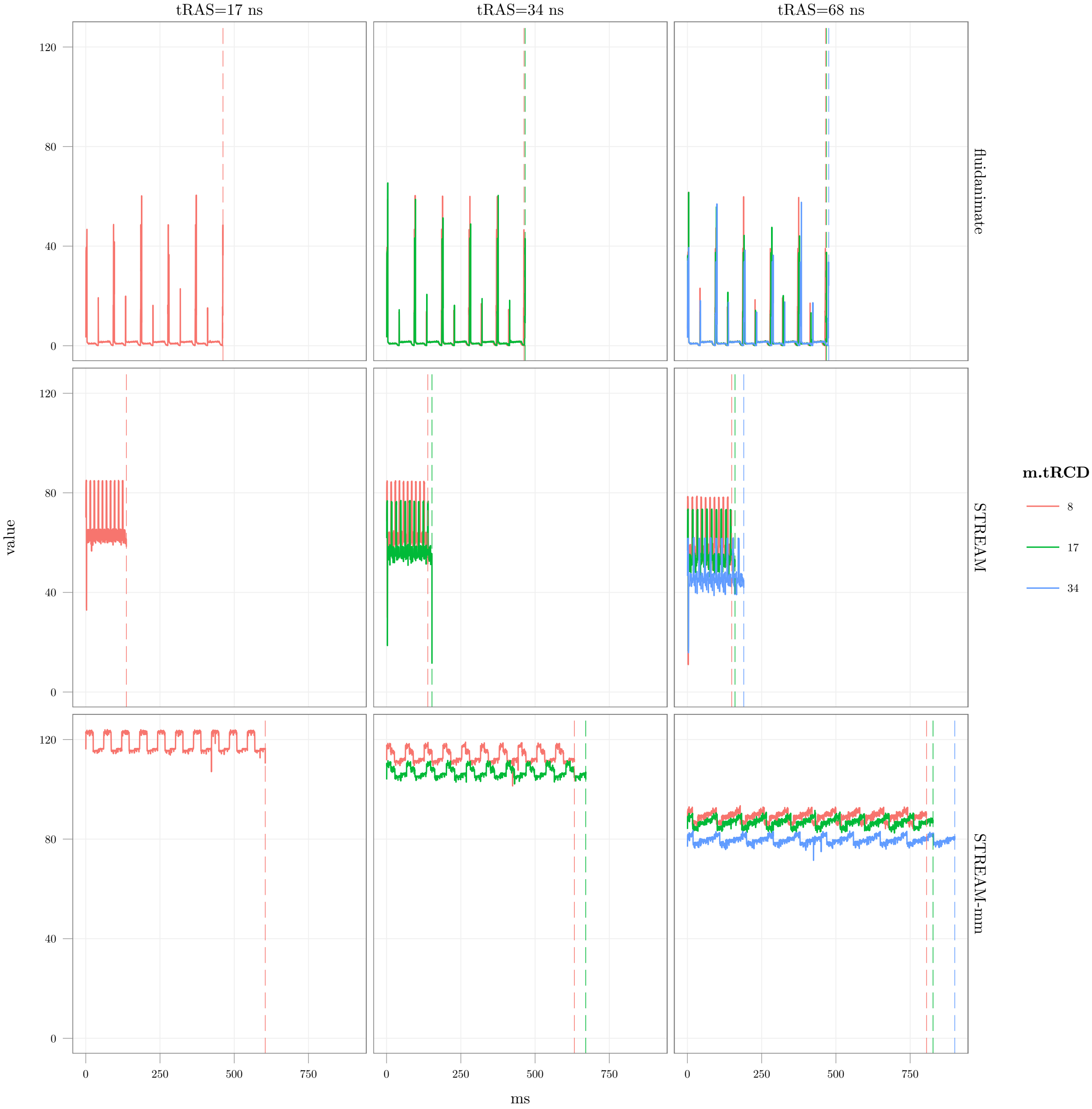


Figure 5.18: Effect of varying t_{RAS} and t_{RCD} on bandwidth over time in a 128 bank HMC.

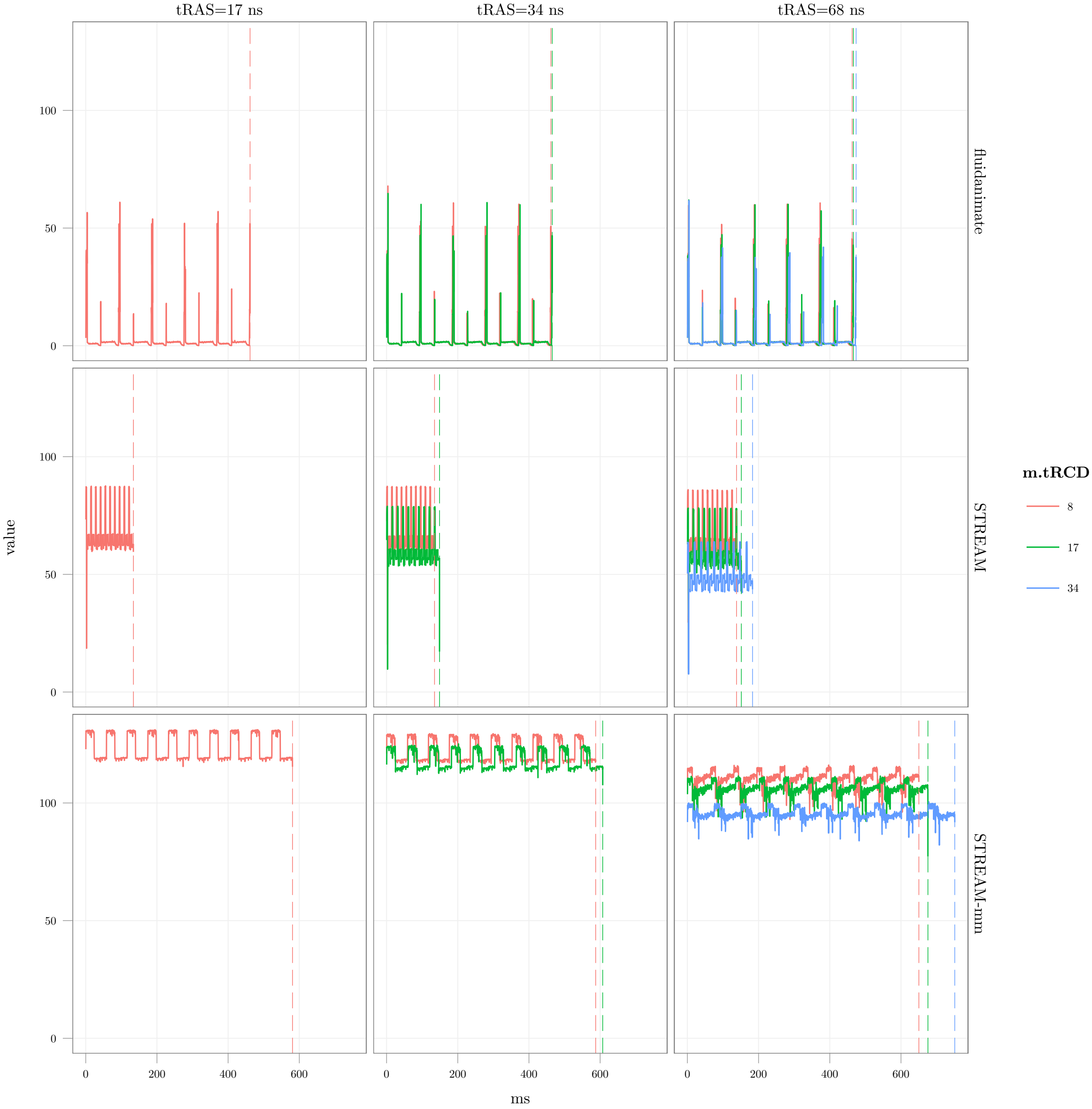


Figure 5.19: Effect of varying t_{RAS} and t_{RCD} on bandwidth over time in a 256 bank HMC.

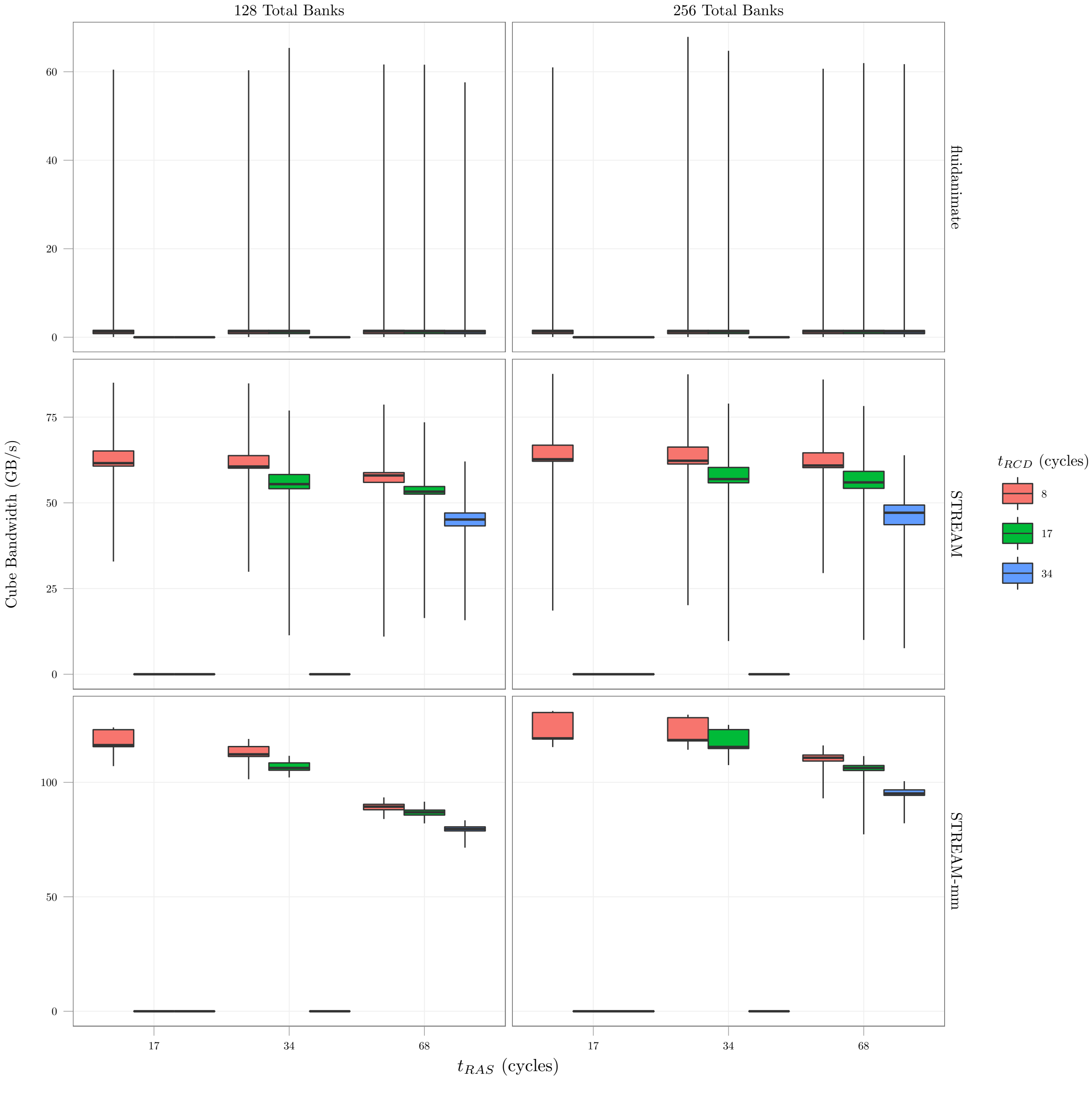


Figure 5.20: Distribution of bandwidths for varying t_{RAS} and t_{RCD} DRAM timing parameters.

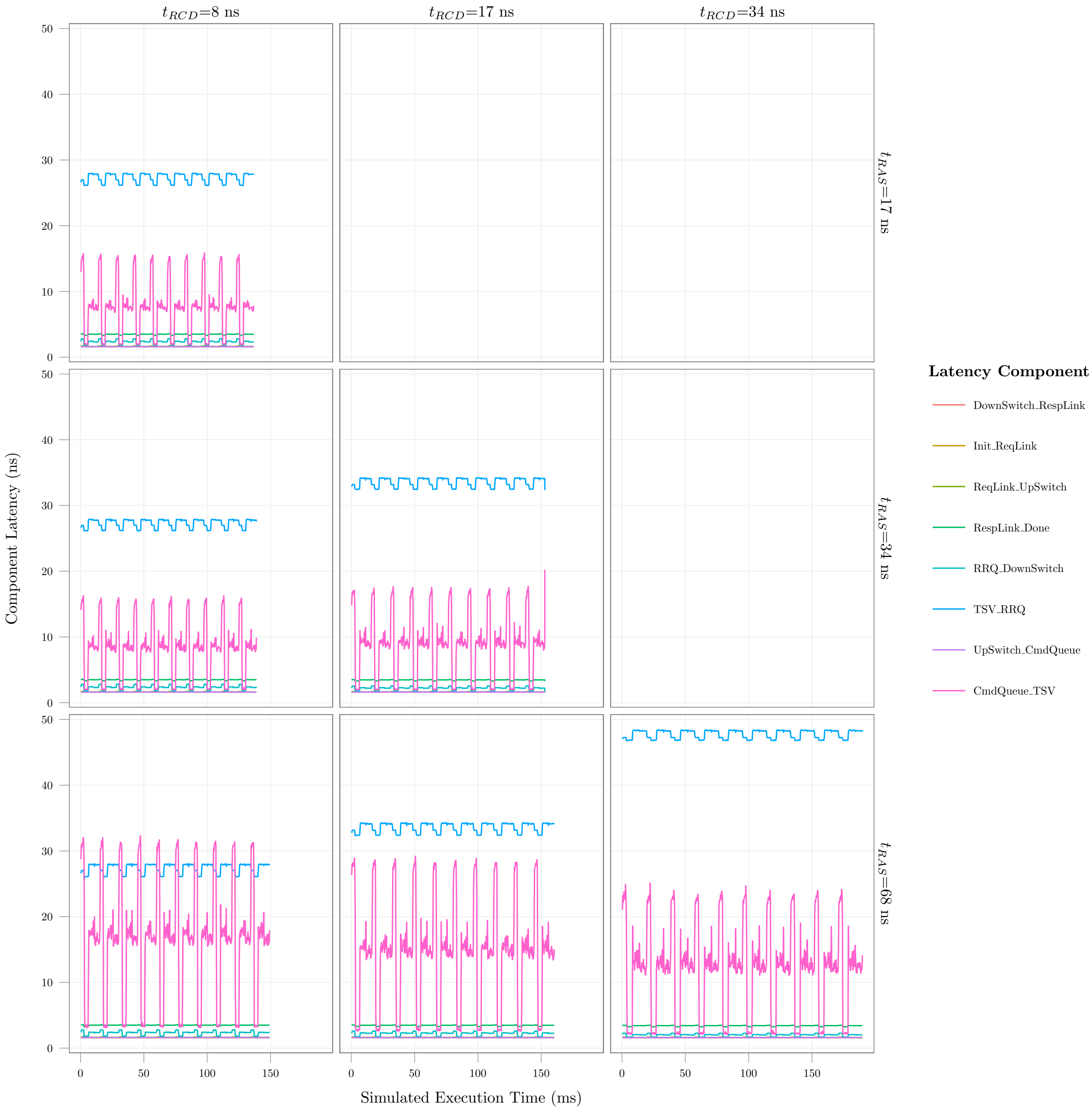


Figure 5.21: Time-varying latency components for various workloads.

burst time). Looking at the plots from top to bottom (t_{RAS} increasing), we see that the blue line stays constant. This is expected since increasing t_{RAS} makes the bank unavailable for a longer period of time, but the data burst begins at the same point in time. The increase in t_{RAS} is seen in the corresponding increase in the pink line that corresponds to the wait time in the command queue. This is because it takes longer for a given bank to start the next request which leads to a longer queueing time in the command queue.

If, however, we examine the plots from left to right (t_{RCD} increasing), we see that the blue line increases steadily. This increase corresponds to the increasing time between the start of a request and the end of the data burst. Note that this latency component also includes the time to burst out the data, so although t_{RCD} doubles, t_{CAS} and the burst time stay constant, so the overall latency does not double.

Finally, we plot the total execution time of all three workloads with the varying DRAM timings (figure 5.22) and see that the impact of the DRAM timings on the execution time is quite small. In the fluidanimate case, the DRAM timings make almost no impact on the execution time because the benchmark, while bursty, is not highly memory bound overall. Even STREAM and STREAM-mm experience only relatively small slowdowns as a result of the different DRAM timings.

Throughout this thesis we use the middle values of 17 cycles and 34 cycles for t_{RCD} and t_{RAS}, respectively. While one might expect that halving both of those timing parameters would drastically improve workload performance, we can see from these graphs that that is not the case. The improved timings have almost no impact on execution time even in a limiting case workload like STREAM-mm.

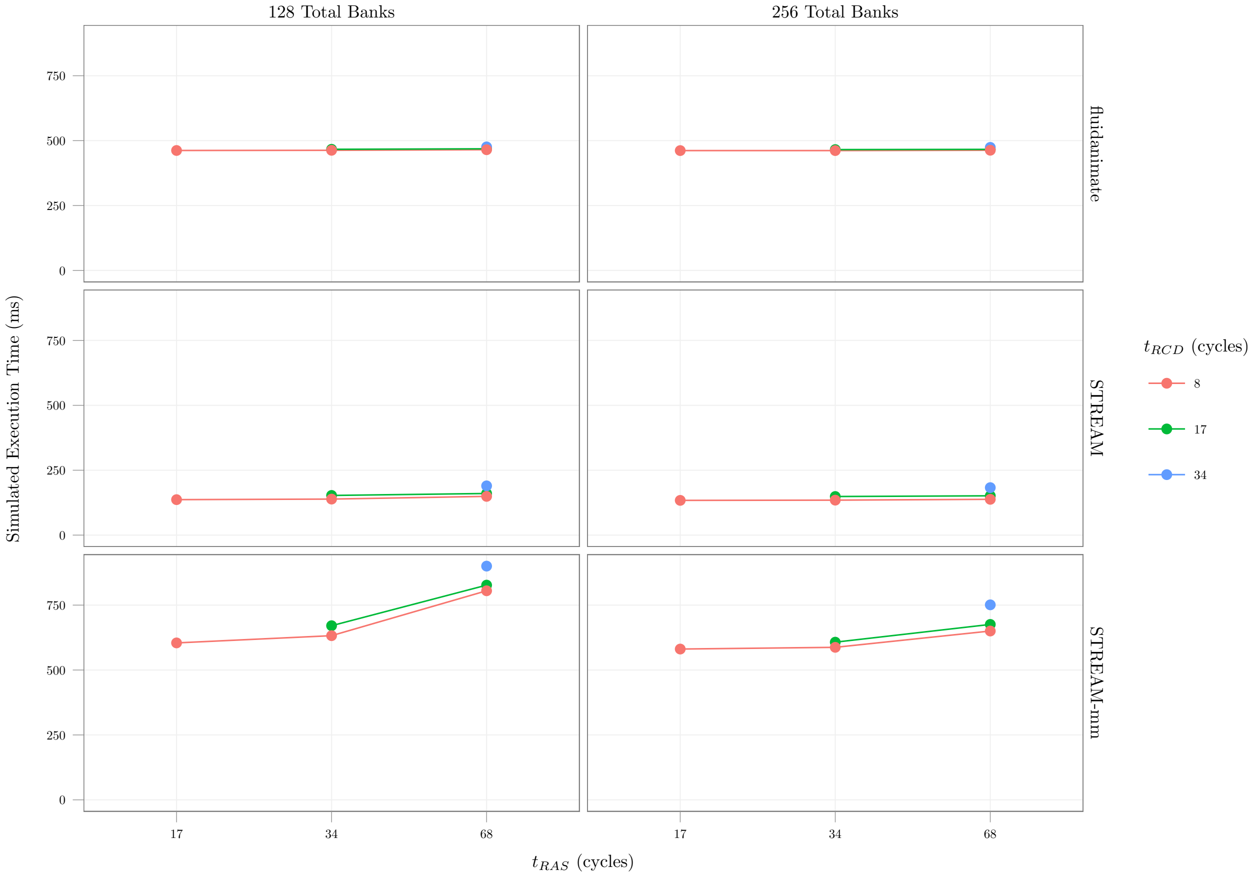


Figure 5.22: The impact of DRAM timing parameters on workload execution time. Even when t_{RCD} and t_{RAS} are both quadrupled the execution time of the workloads is largely unaffected even for highly memory intensive applications such as STREAM-mm.

This can be explained by fact that there is so much bank-level parallelism in the cube that, on average, any individual bank is not able to slow down the system. In fact, for a closed page system, we can compute the ideal bank bandwidth by simply taking $\frac{size_{burst}}{t_{RC}}$. In the 128 bank configuration with the worst DRAM timings in this experiment, we get a per-bank bandwidth of 0.94 GB/s. Multiplying this number by the number of banks in the cube yields an overall DRAM bandwidth of 120 GB/s. This means that even with slow DRAM devices, an HMC can still achieve high overall throughput. Figure 5.22 supports this hypothesis because the 256 bank configuration suffers a much smaller slowdown with degraded DRAM timings as compared to the 128 bank configuration.

5.8 Address Mapping

One of the features presented in the HMC specification is the ability to customize the address mapping scheme. Address mapping is the process by which a physical address's bits are used to select which specific resource in a particular memory device will be used to service that request. A suboptimal address mapping scheme can cause memory requests to continuously cause conflicts to shared resources and degrade performance. On the other hand, an optimal address mapping scheme can spread requests evenly and utilize the memory system's full parallelism. This makes the address mapping scheme a vital consideration for performance. However, the bad news is that address mapping will always be workload specific: an address mapping that is ideal for one workload might cause performance degradation for another workload due to a difference in access pattern over time.

As discussed in [53], the address mapping scheme for a closed page system should avoid bank conflicts by spreading requests from neighboring cache lines as far away as possible. For a DDRx system this means mapping the channel to the least significant bits followed by bank then rank. Translating this mapping scheme to the HMC would mean putting the vault address in the lowest bits followed by bank then partition. For a closed page buffer policy, the row and column addresses are effectively inconsequential since all rows are closed immediately after the request is completed. In essence, a closed page address mapping scheme only need concern itself with what request maps to what bank. For this reason, all of the address mapping schemes chosen have the column and row addresses in the top-most bits so as to leave the lower-order bits that have lower locality to generate request spread.

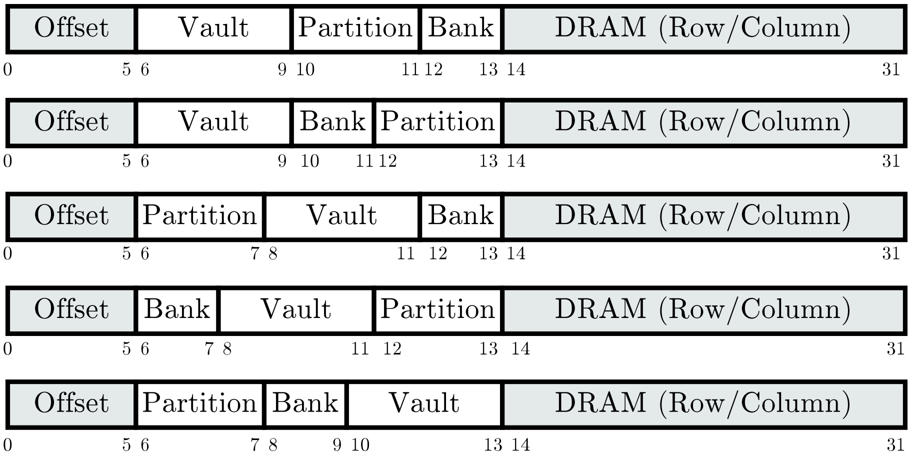

Offset | Vault | Partition | Bank | DRAM (Row/Column)
0 5 6 9 10 11 12 13 14 31
Offset | Vault | Bank | Partition | DRAM (Row/Column)
0 5 6 9 10 11 12 13 14 31
Offset | Partition | Vault | Bank | DRAM (Row/Column)
0 5 6 7 8 11 12 13 14 31
Offset | Bank | Vault | Partition | DRAM (Row/Column)
0 5 6 7 8 11 12 13 14 31
Offset | Partition | Bank | Vault | DRAM (Row/Column)
0 5 6 7 8 9 10 13 14 31

Figure 5.23: Single cube address mapping schemes

The HMC specification defines the default address mapping scheme as the offset in the least significant bits, followed by the vault address, bank address, and DRAM address. The details of the DRAM address are not fully specified since it is unknown which bits of the DRAM address select the partition, row, and column. We select five different address mapping schemes shown in figure 5.23 and execute them in MARSSx86 running eight threads on eight cores attached to a 256 bank/16 vault HMC with 240 GB/s aggregate link bandwidth and 160 GB/s aggregate TSV bandwidth. We choose five workloads based on the characterization of memory bandwidth presented in section 5.6.2: these workloads generate high bandwidth to memory either in a sustained or bursty fashion.

We denote our address mapping schemes as a series of fields which represent parts of the address from the least significant to the most significant bit. As previously stated, we assume a 64-byte cache line size and, for simplicity, we force all addresses to be aligned to a cache line boundary[3]. This means that the bottom six bits of the address will be zeros which correspond to the offset and low bits of the column address (i.e., $2^6 = 64$ bytes per transaction). Since this section only considers single cube configurations, there are zero cube bits present in the addresses.

5.8.1 Single Cube Address Mapping Results

First, we examine the bandwidth time-series of each of the five workloads under each of the five address mapping schemes as shown in figure 5.24. In order to make the relative performance of each scheme clearer, we annotate the graphs with a vertical line that represents the final time step of each simulation. Since each execution runs between the same two points in the program, a vertical line further to the left indicates a shorter total execution time.

[3] While this assumption is acceptable for cache line fills, it is not valid for DMA requests

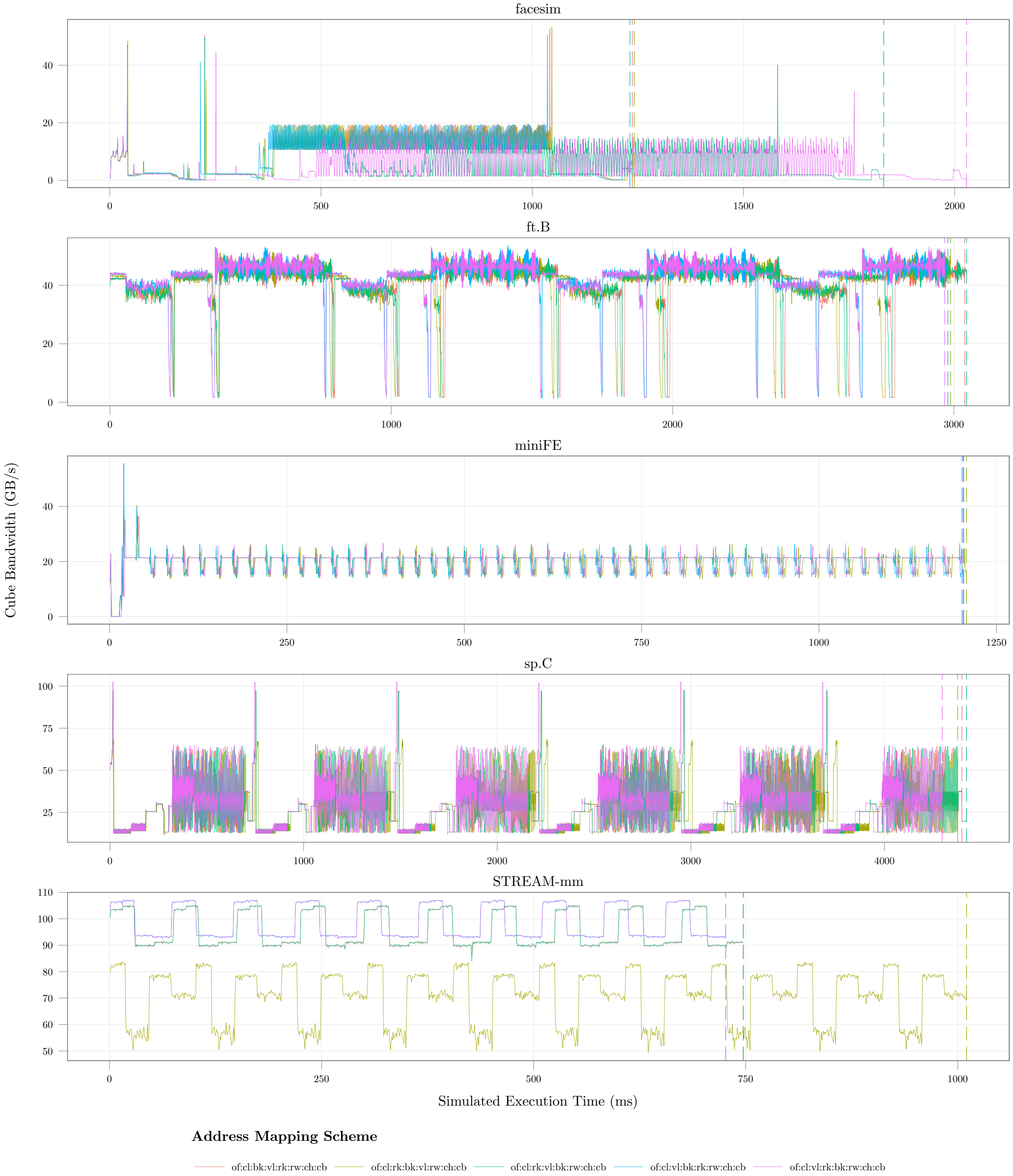


Figure 5.24: Performance of various workloads under various address mapping schemes with a single cube. The dashed vertical line at the end of each simulation indicates the ending point of the workload's execution. A vertical line farther to the left indicates a shorter execution time.

Examining the completion times of the various workloads under different address mapping schemes, we see that the blue line that corresponds to the vault:bank:partition address mapping scheme results in the best performance and lowest execution time among these workloads. If we consider a vault to be equivalent to a DDRx channel and a partition to be equivalent to a DDRx rank, then we see that this mapping scheme corresponds exactly to the optimal closed page mapping scheme described in [53]. The reasoning is fairly straight forward: since a closed page policy assumes a stream with little or no locality (spatial or temporal), then it is best to distance adjacent cache lines by spreading them among the most independent elements (i.e., vaults). Putting the partition bits higher than the bank bits allows the controllers to minimize bus switching time (i.e., reads to different partitions that incur a one-cycle turnaround penalty). Furthermore, placing the bank bits lower in the address reduces the chances of a bank conflict within a given partition.

If we look at the other mapping schemes, there are some notable exceptions to the heuristic of putting the vault bits in the lowest part of the address. In particular, when the partition:bank:vault mapping scheme is paired with facesim from the PARSEC Benchmark suite, it performs nearly as well as the optimal vault:bank:partition scheme. However, for three out of four of the other benchmarks (miniFE, sp.C, and STREAM-mm) the partition:bank:vault scheme has the worst performance. This is especially true of STREAM-mm, which suffers a particularly large slowdown as a result of using this mapping scheme. Furthermore, unlike the other workloads, facesim performs most poorly under the vault:partition:bank scheme which has the vault address mapped to the low bits. Changing the ordering of the partition and bank positions causes a vast difference in performance.

To better visualize the effects of address mapping, we examine how well the address mapping schemes spread requests between banks over time. As mentioned previously, we are most concerned with bank-level request spread in a closed page system: an optimal closed page address mapping scheme should utilize all banks evenly without generating hotspots to any particular bank. In order to collect this data, we instrument the simulator to output the number of cycles per epoch in which bank has an open row. This information is collected for each bank in the system for each epoch and is displayed as a heat map: the bank's linear index (0-255) is along the y-axis and the simulated time is along the x-axis. The heat map colors range from black (underutilized) to red (half-utilized) to yellow (fully utilized). To make the heat maps easier to read, each value is scaled on a per-workload basis (i.e., if a workload only ever utilizes a bank half of the time, the highest yellow value corresponds to 50% utilization; if another workload has a maximum utilization of 90%, the most yellow value corresponds to 90% utilization). Note that this scaling makes it possible to compare address mapping schemes for a single workload by looking at the relative coloring, but the colors **cannot** be compared between workloads.

Figure 5.25 shows the HMC bank-level utilization heat maps for the STREAM-mm benchmark's execution. Simply glancing at the heat maps, one can get a fairly good idea of which mapping schemes perform well: horizontal bands represent bank hotspots that persist over time. Vertical stripes can be the result of the CPU's main memory request rate and so are not inherently related to the address mapping

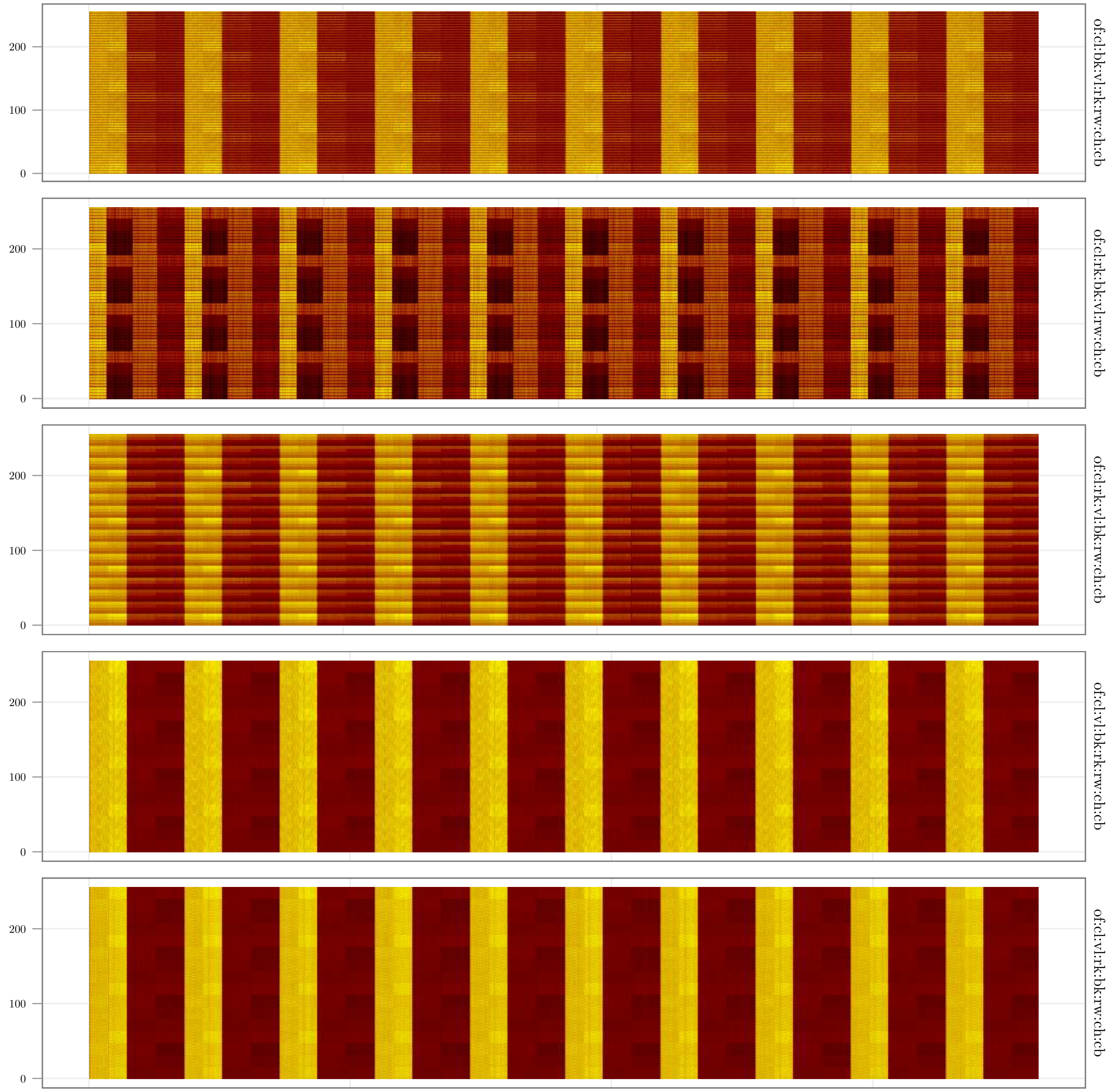


Figure 5.25: Heatmaps for five address mapping schemes for the STREAM-mm workload over time

scheme. However, if a vertical band has many different colors within it, this is an indication that the mapping scheme is performing poorly (i.e., since this is an indication of some banks being over-utilized while others are idle). Since the values are normalized per workload, the relative brightness of the entire heat map is also a good indication of performance.

The mapping scheme with the vault bits in the upper part of the address has the most color variation with many dark spots and horizontal bands. We zoom in on the first part of the execution time and compare the two heat maps along with their bandwidth graphs (see figure 5.26) Indeed we find that it has significantly lower bandwidth than the schemes where the vault bits are in the lower portions of the address.

sp.C has areas of intense memory access and then long periods of very low memory bandwidth. 5.29 shows a comparison of two address mapping schemes for a very short duration of time during a bandwidth spike. The impact of address mapping can clearly be seen here in that the horizontal striping of the rank:bank:vault address mapping scheme extends the duration of the access time. The vault:rank:bank scheme, however, accesses all of the banks nearly evenly and the duration of the spike is much smaller.

5.9 Memory Performance Comparison

When discussing HMC limit case simulations and full system workloads that can consume over 100 GB/s of bandwidth, it is easy to forget that this is not the

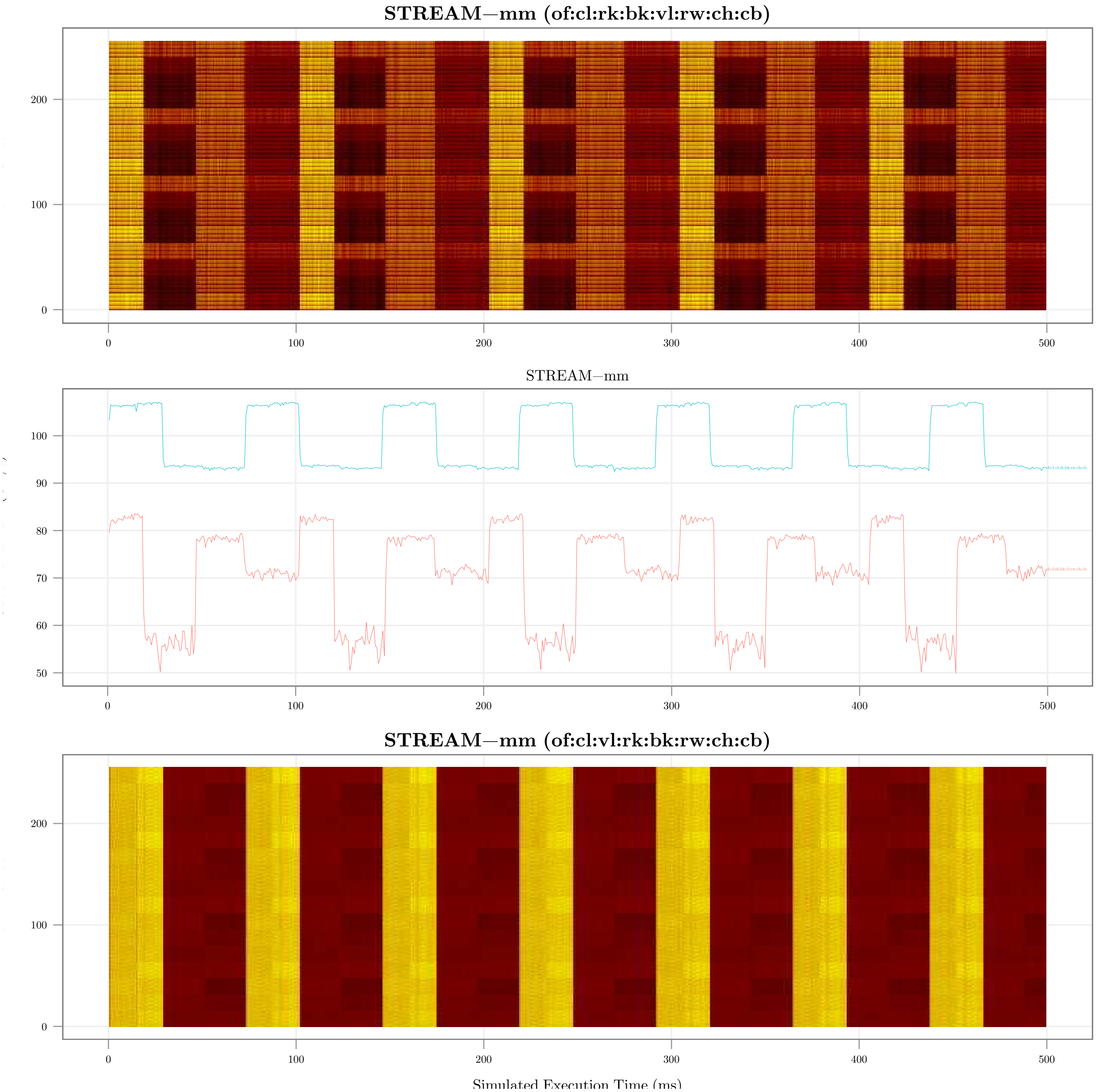


Figure 5.26: Address mapping scheme comparison for the STREAM-mm workload. Dark spots indicate banks with low utilization whereas red and yellow spots represent banks of high utilization. Horizontal stripes are an indication of a mapping scheme that hot spots banks, and thus is suboptimal.

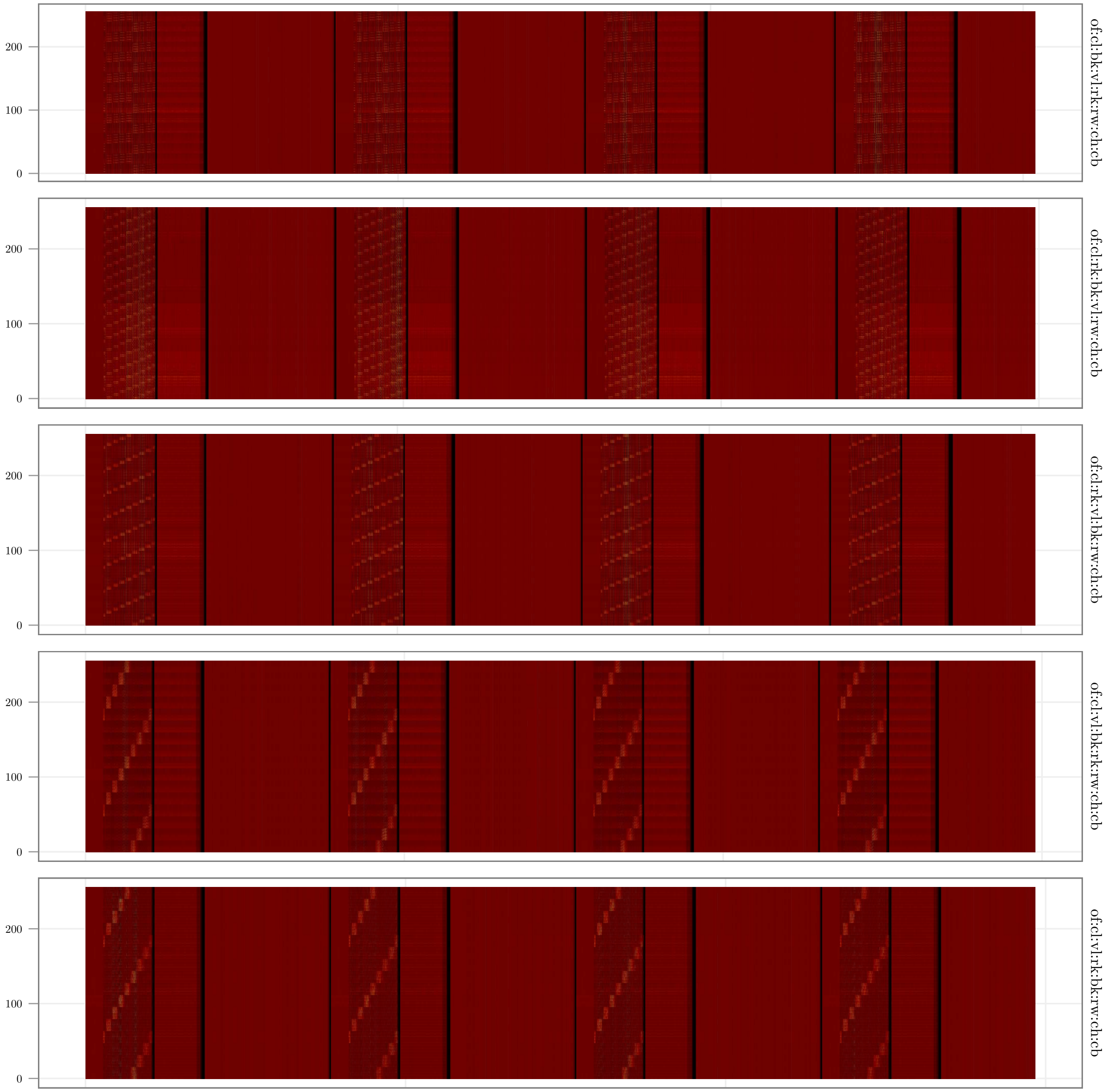


Figure 5.27: Heatmaps for five address mapping schemes for the ft.B workload over time

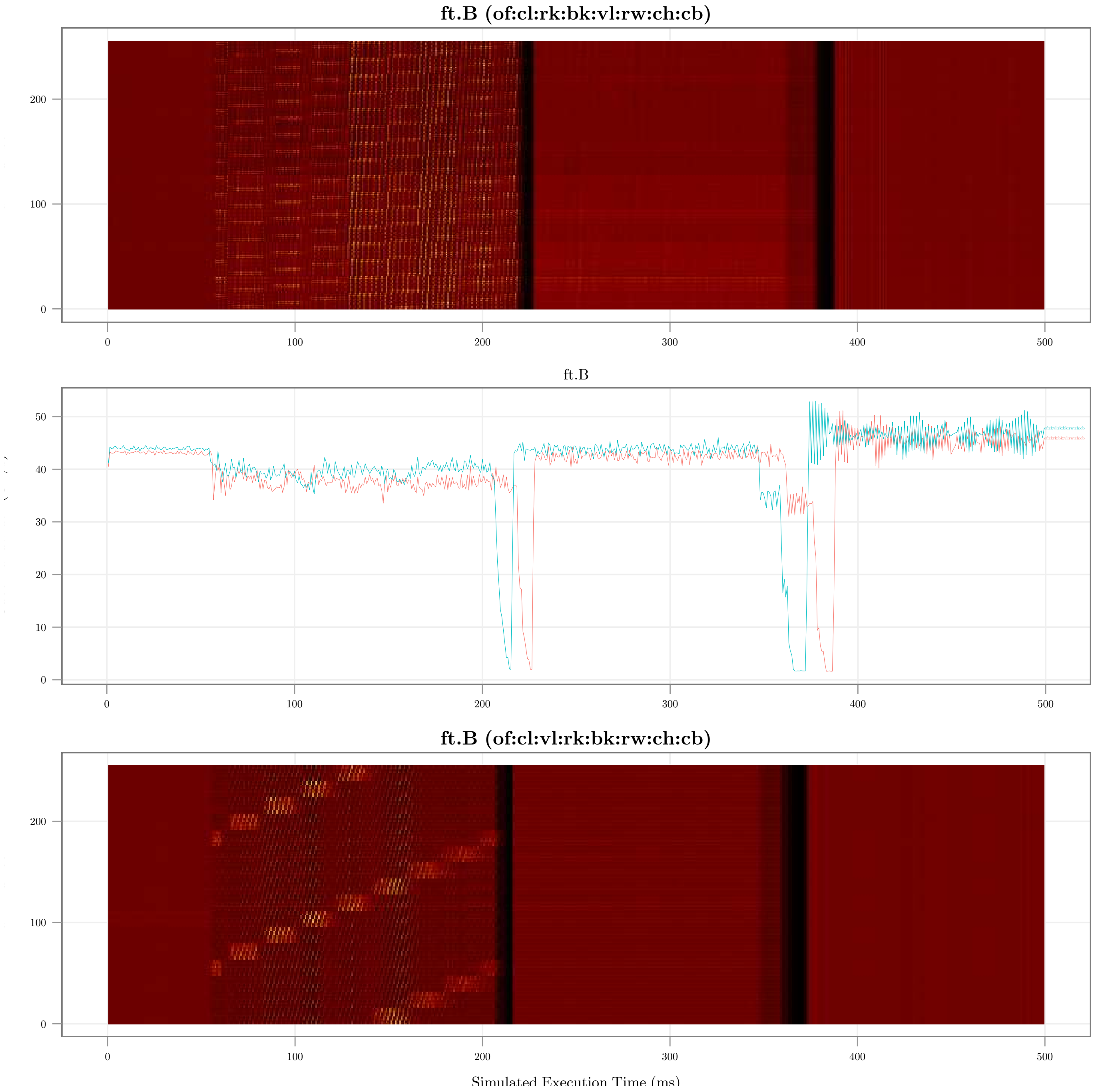


Figure 5.28: Address mapping scheme comparison for the ft.B workload. Dark spots indicate banks with low utilization whereas red and yellow spots represent banks of high utilization. Horizontal stripes are an indication of a mapping scheme that hot spots banks, and thus is suboptimal.

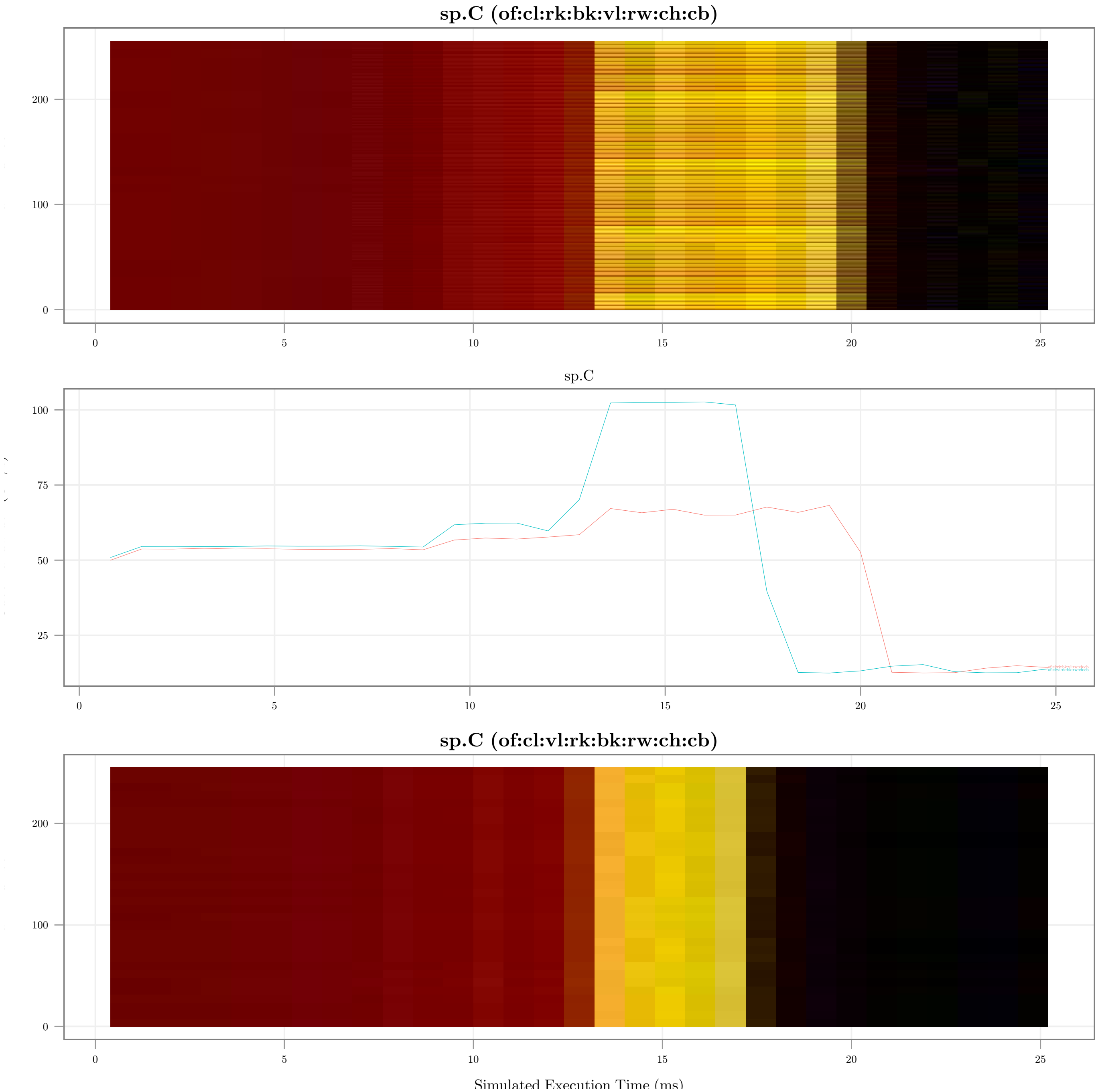


Figure 5.29: Address mapping scheme comparison for the sp.C workload. Dark spots indicate banks with low utilization whereas red and yellow spots represent banks of high utilization. Horizontal stripes are an indication of a mapping scheme that hot spots banks, and thus is suboptimal.

status quo for current memory systems. That is, current generation DDR3-1600 parts have a theoretical maximum throughput of 12.8 GB/s per channel. Each such channel requires hundreds of CPU pins and a memory controller that sits on the CPU die, far away from memory. To put the performance of the HMC into perspective, we configure an aggressive quad channel DDR3-1600 and simulate it in DRAMSim2 (see section 4.4.3). Additionally, we simulate the perfect memory model to give a sense of how memory intensive the workloads are.

Figures 5.30 and 5.31 show the main memory bandwidth of several workloads over time. To reiterate, since these workloads execute between the same two points in the program, it is possible to calculate the speedup by comparing the relative lengths of the lines (i.e., shorter line means lower execution time). Upon examining these graphs, one immediately notices that the HMC memory system can only make an impact for the workloads with the highest memory intensity. That is, for workloads such as fluidanimate and miniFE in figure 5.30, the usage of an HMC memory system fails to make any significant impact on the execution time. In these cases, even an unattainable "perfect" memory system can only provide a reasonably small improvement.

For memory intensive workloads such as STREAM, however, the HMC can provide a significant speedup simply by changing the main memory system. The bottom graph of figure 5.31 shows that the DDR3-1600 system is constantly running near its theoretical peak bandwidth, but the HMC system still manages to complete the workload 2.1 times faster. This is a drastic departure from the performance of current generation memory systems, especially if the energy efficiency of the

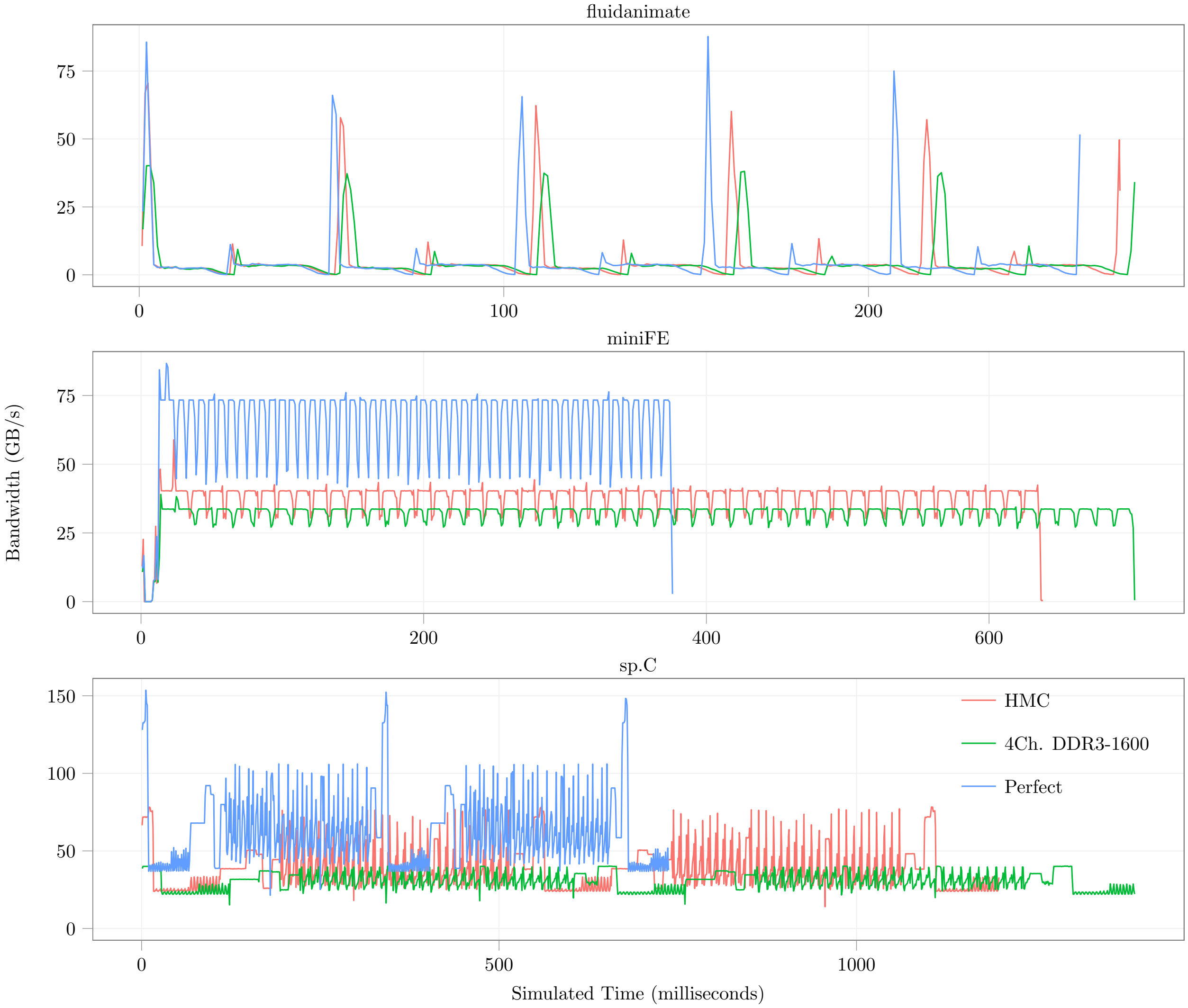


Figure 5.30: Comparison of memory system technologies: Quad Channel DDR3, HMC, and perfect. (1)

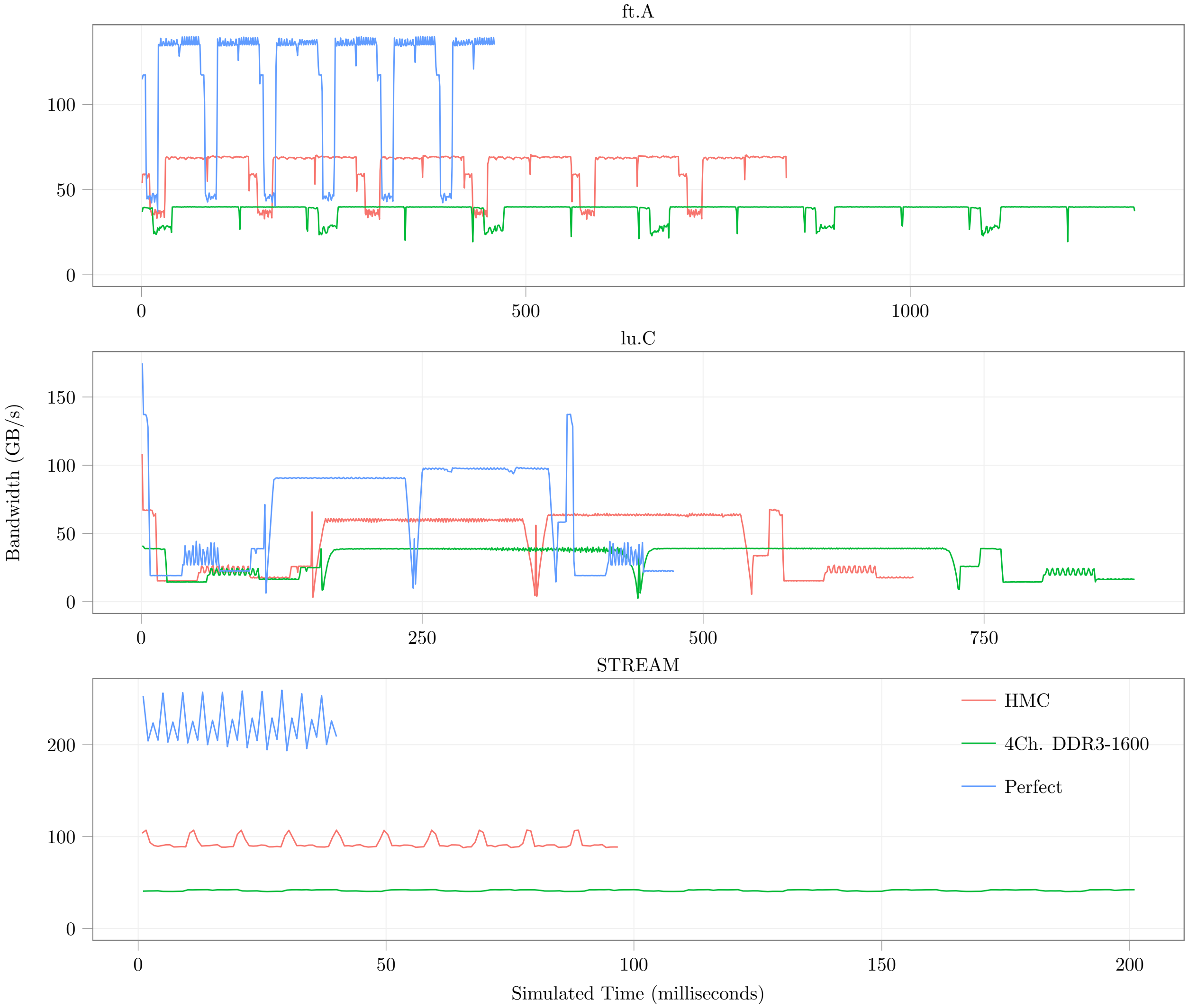


Figure 5.31: Comparison of memory system technologies: Quad Channel DDR3, HMC, and perfect. (2)

commercial devices match current predictions (see section 2.2).

Chapter 6

Multiple Cube Topologies

6.1 HMC Chaining Background

One of the unique features of the HMC is the ability to create nearly arbitrary cube topologies by linking multiple cubes together. This feature is enabled by several different features of the HMC: a general purpose serial link interface, the presence of general purpose logic within each HMC device, and multiple input and output links per device. A traditional DDRx memory system forms a uniform memory space with no hierarchy because of the use of a single broadcast bus for all memory devices per channel. The memory devices always operate as slave devices as they have no general purpose logic on the DIMM. Using multiple CPU sockets introduces an element of non-uniform memory access (NUMA) when each processor socket is connected to its own local DRAM while being able to access the DRAM connected to any other socket by traversing an interconnect such as Intel's QuickPath Interconnect (QPI). While this can be considered a simple topology of sorts, each memory channel still requires a CPU to act as a pass-through device.

The HMC, in contrast, contains general purpose logic at the base of each cube which replaces the CPU as the pass-through device. Furthermore, since the links transmit data using a general-purpose protocol, the HMC links can be thought of as analogous to the QPI interface which connects CPU sockets. In some sense, the

HMC's support for chaining is isomorphic to the use of multiple sockets with local and remote memories, but can be achieved without the use of extra CPU sockets. The use of multiple CPU sockets is orthogonal to the use of the HMC memory system and so it is possible to have a second level of NUMA: a local HMC topology where some portions of the address space are in cubes that are multiple hops away and a remote HMC topology connected to a different CPU socket. The HMC specification allows for one cube to receive requests from multiple hosts and some previous work has also examined the possibility of using a set of HMCs to function both as the memory system as well as the processor interconnect [41].

The HMC specification explicitly discusses the creation of cube topologies, but does not discuss many specific topologies or their design implications. According to the document, unlike many network applications where each router in the network can dynamically update routing tables, requests are source-routed (i.e., the host determines a route for a request and it is not changed while the request is in flight). The specification also provides a brief explanation that each HMC is loaded with vendor-specific topology information which is used to route the request. No additional implementation details are provided by the specification. Finally, the specification states that a response to a request must return along the same path as the original request.

Since creating cube topologies is likely to be a common and important use case of the HMC, our goal is to take a look at some potential implementations and understand the trade-offs involved. In this section, while we will try to keep a realistic view of what cube topologies might look like, we will modify some of the assumptions proposed in the specification to examine the impact of some more interesting cases. We will maintain the requirement that responses must proceed along the same route as the corresponding request and that the requests are to be source-routed. However, in some cases, we will assume that the host controller has access to information about the congestion level within the topology. This is a realistic assumption given that the host determines all routes. Therefore, it is fair to assume it could store information about which routes are in use by outstanding requests. We will also assume that cubes will not send requests "backwards" through the network to form cycles as this is unlikely to generate useful routes and complicates route generation.

The HMC simulator is built on a set of primitives which can be composed to form arbitrary topologies. In this section, we will consider two basic topologies: a linear chain and a ring (which is simply a chain that wraps the final cube's links back to the CPU).

6.2 Routing Background

One of the challenges in building different HMC topologies is the need to generate possible routes. The HMC specification implies that the routing information will be generated by the host and programmed into each cube as a lookup table: a request arrives at a cube, the header information is used to index the lookup table, and the packet is sent along to either a local vault or out to a pass-through link. In simulation, however, enumerating the set of all possible routes to a given destination is challenging due to the number of possible routes in certain topologies. Trying to generate a lookup table by hand would be both time-consuming and error prone. Furthermore, trying to re-generate such a lookup table each time one wanted to test a new topology or change the number of cubes would be intractable.

In order to avoid hand-writing the routing information for each topology, the simulator automatically generates these routes by recording all paths between a given host link and a given vault and storing a routing table at each host link. The route generation is done in two steps. First, a breadth-first search is performed starting from each host link and terminating upon reaching all vaults. As the search proceeds, each transmitter and receiver is assigned a rank equal to the number of hops from the host link transmitter. After the nodes have been ranked, the second step performs a depth-first search starting from each link and recording every path that ends in a vault. This DFS is restricted to only consider a successor node whose rank is higher than the current node. In essence, the initial BFS step performs a topological sort on the nodes to prevent cycles in the depth-first search.

Once all possible routes from each CPU link to each vault are found and recorded at the CPU, the routing table can be queried to produce a list of all possible routes that reach the target vault from a given host link. However, once the host is equipped with this information, a second question arises: what heuristic should be used to pick a route for a given request? In some topologies, such as a linear chain, all of the routes to a given vault have the same number of hops, but given that there are $\frac{l}{2}$ pass-through links for each cube (where l is the number of links per cube), there is a non-trivial number of routes available to choose from for a moderately sized chain (i.e., $(\frac{l}{2})^n$ where n is the number of cubes). Other topologies (such as a ring) can reach a target vault from several different paths which have different numbers of hops. Some heuristic might be necessary to choose whether it is worth it to choose a longer but less congested path over a shorter but busier path.

In both the linear chain and ring, all vaults are accessible from any link. When discussing the order in which to consider routes, only routes that lead to a target vault are considered for selection.

6.3 Route Selection Algorithms

For a source routed system like the HMC, the host only has access to local information to make routing decisions. This information must be available either from the request itself (e.g., the address of the requests) or from some local data stored at the host. The route selection algorithm really must answer two largely independent questions: which link will this request be transmitted on and which route will it take to get to the target vault.

The first question has to do with the reachability of the target vault from a given link (i.e., in some configurations, not every link may be able to reach every vault) and the available resources at a given link (i.e., a link may not have buffer space for a write request that takes up 80 bytes but may have space for a 16 byte read). The second question has more to do with optimization. That is, once the host knows that a given link can reach the target vault and that link has the buffer space to accommodate that request, any route will satisfy the request but some may be better than others.

The simulation, therefore, splits the problem into two phases. First, a "link chooser" picks a link which can reach the vault and has buffer space, and then a "route chooser" looks at the available routes from that host link and chooses one of those. That route is encoded into the request and is never changed. As mentioned previously, we enforce the condition that a request must reverse its original path on the response path.

6.3.1 Link Choosers

We will briefly outline the various algorithms for choosing a link in this section.

6.3.1.1 Random

The random link chooser simply enumerates the list of available links that reach the target and picks one based on a uniformly distributed pseudo random number generator. It serves mostly as a control to show what kind of results can be achieved with no real heuristic at all.

6.3.1.2 Address-based

The address-based selection scheme uses the address of the request to choose an outgoing link. That is, we select the low order bits just above the transaction offset and use those bits to generate a link number. If the selected link is busy, it must wait until that link frees up before the request can be transmitted. The low order bits of an address stream are likely to look fairly random but unlike the previously described “random chooser”, this algorithm will make a request wait until the specified link frees up even if other links are idle.

6.3.1.3 Buffer Space

The buffer space selection scheme tries to pick a link that has the most available buffer space. The rationale is that the buffer space usage is roughly proportional to the link utilization at any given point in time. Therefore, choosing the link with the most buffer space is equivalent to choosing the least busy link to ensure an even spread of requests among links in the long run.

6.3.1.4 Read/Write Ratio

As discussed previously, the maximum theoretical efficiency of the links is dependent on the read/write ratio. Because of this feature, it stands to reason that to make the most efficient use of the links, the link heuristic should try to maintain the read/write ratio for any given link as close to the optimal point as possible. Since we only consider 64 byte requests, we have shown that the optimal read/write ratio is 56% reads. Therefore, this chooser keeps a per-link counter of the number of reads and writes issued to that link and chooses to send the read or write to the available link that will push the read/write ratio toward the optimal. That is, if a link is below the optimal read/write ratio, the algorithm will prefer to send reads to that link (and the opposite for writes). Although read/write ratio is not the only determinant of performance in the system, it is a reasonable approach given only local information.

6.3.2 Route Choosers

After a link has been selected by the link chooser, a second step of the algorithm must decide on the route that the request will take through the entire memory system. Implementing a heuristic to choose a route is somewhat more difficult given the constraint of source routing. A typical network routing algorithm can dynamically change the route of a packet at each node in the network. Therefore a path can change dynamically to become more optimal since the congestion information is distributed throughout the network. However, with a source routed algorithm, the host controller can only use local information that is available either from the request itself or from historical information stored locally.

The input to the route chooser is a set of routes that originate at the chosen link and terminate at the vault number of the request.

6.3.2.1 Random

The random route chooser simply chooses a route at random from the list without considering any properties of the routes. As with the link chooser case, this case is meant to demonstrate the performance when there is no selection logic at all.

6.3.2.2 Round Robin

The round robin heuristic keeps a history table of which route index was selected on a previous request between the given link and the given vault. While the amount of logic to implement this scheme is very simple, it does require the controller to keep a static history table of pointers to the routing table that has a size of $N_{links} * max(N_{vaults})$. This might be problematic in two ways: (1) if there is an increase in cube density, the controller may need more entries, which limits expandability of the system; (2) for large configurations with many cubes, the size of this table may become significant.

6.3.3 Congestion Aware

The congestion aware heuristic tries to emulate the "ideal" case for a routing method which is to keep enough state about the traffic flowing globally through the system to be able to make an informed decision about what route will encounter the least congestion. This heuristic is the only one that can make a reasonable choice in a ring topology as it will sometimes choose a longer but less congested path over a shorter but congested path. In the simulator, this strategy works by examining the number of flits waiting in the buffers along each route and choosing the smallest sum. Since the number of buffer entries are roughly proportional to the latency that the request will encounter, this is a reasonable indicator of the path congestion.

This scheme could be implemented in hardware, but it would require a large amount of logic and storage to work. Upon choosing a route for a given request, the

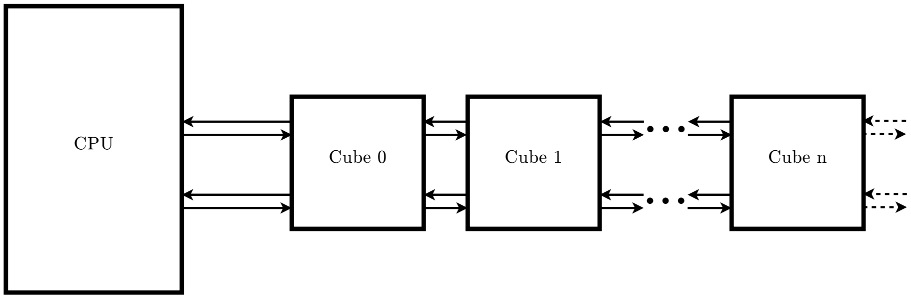


Figure 6.1: A linear chain topology

controller would have to examine which queues would be utilized by that request and increment a counter for each one. As responses flow back to the controller, the counters for the utilized queues would be decremented. Although this would allow the controller to utilize this scheme, it would impose a limit on scalability as a single extra cube has many buffers inside, which would require the table to grow significantly. The logic to examine this table would also likely be too slow and expensive.

While it is unlikely that this technique would ever be implemented in an actual system, it serves as a good comparison point of how an "ideal" routing strategy might perform and how big of a performance impact the routing strategy has.

6.4 Topologies

6.4.1 Chain

The simplest topology that one could implement for a network of cubes is a simple "linear chain" shown in figure 6.1. Half of a cube's links are connected to the left and half are connected to the right. The half of the links in the final cube

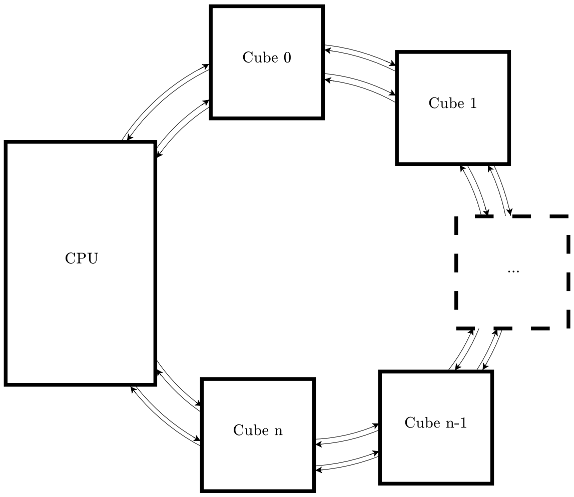


Figure 6.2: Block diagram of a ring topology. The CPU uses twice as many links as a linear chain and halves the distance to the farthest cube.

are left unconnected and are powered down.

While a linear chain is a good way to increase the system's capacity, it does have some drawbacks. First, it is clear that the request latency increases significantly as a request moves farther down the chain. Furthermore, the cube closest to the CPU will likely be overloaded as it must not only service local requests, but transmit passalong requests and responses for all of the other cubes. To make matters worse, even though the first cube in the chain must transmit lots of extra data to and from the CPU, it is only connected to the CPU with half of its links.

6.4.2 Ring

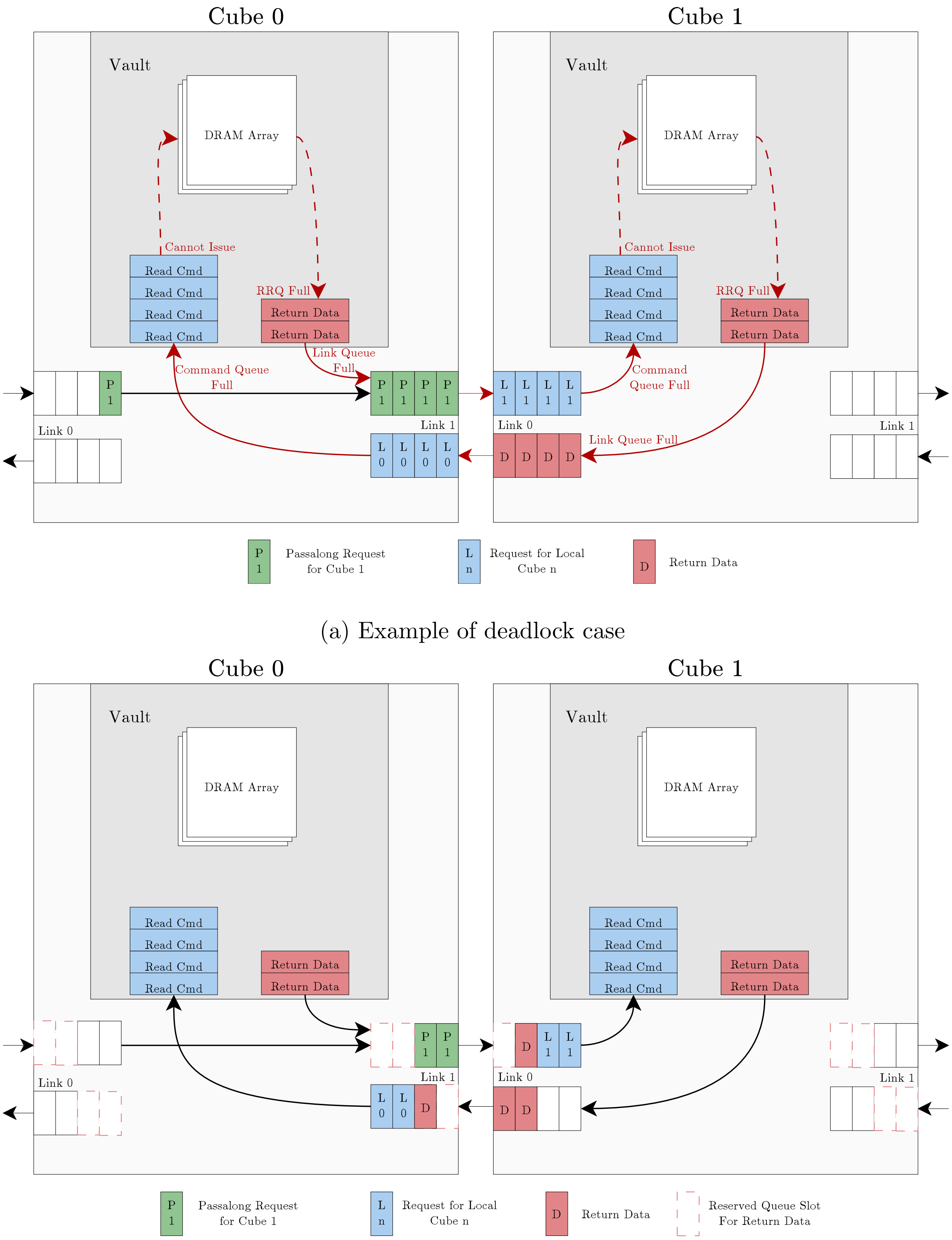
Some of the problems of the linear chain topology can be ameliorated by making a simple modification to the topology. By connecting the two unconnected links at the end of the linear chain back to the CPU, the topology becomes a ring instead of a chain. This fixes two of the major drawbacks of a linear chain. First, while the maximum path length remains the same between the two topologies, a ring reduces the average path length by a factor of two. That is, because each cube can be reached by traversing the ring either clockwise or counterclockwise, there is always a route to a cube that has a maximum path length of half the number of cubes. The second major benefit is that a ring allows the CPU to communicate with the cubes using twice as many links as the linear chain.

Although these benefits make the ring seem like a natural choice over a simple chain, the ring topology does introduce a problem. Since requests and responses can travel both clockwise and counterclockwise around the ring, both passalong packets and local requests/responses intermix in the link queues. This can lead to the situation shown in figure 6.3a. Responses travelling clockwise and counterclockwise can create a cyclic condition where two vaults become deadlocked. Neither one can issue new requests due to a full response queue, but passalong packets are stuck behind incoming command packets. The cycle formed by the requests is highlighted by the red lines in 6.3a.

Triggering the deadlock requires very high load on the memory system that can potentially cause several buffers in the system to fill with certain types of requests. In the random stream simulations performed in section 6.5, this deadlock condition occurred in less than 2% of simulations. The “congestion aware” heuristic described in section 6.3.3 would be most likely to trigger a deadlock condition. This is because this heuristic has a tendency to fill any unused buffer space in the memory system leading to a situation where a deadlock is more likely. Due to the complexity of the data paths in the system, figuring out the source of the deadlocking was quite challenging.

This deadlock condition can be overcome by making two observations about the nature of the requests and responses. The first observation is that response packets can always make forward progress because their final destination is always a CPU and the CPU will always eventually accept the response packet. The second observation is that the deadlock in figure 6.3a can be avoided if the requests from the vault's read return queue can drain its responses back into the cube network. Therefore, if we keep space reserved in the network specifically for responses, the system can always make forward progress (see 6.3b). Some reserved response slots will always become available due to the CPU consuming responses meaning that the vault can always eventually return responses and consume more requests.

Another way to overcome the deadlock condition is to prevent the situation where requests and responses can flow in both directions around the ring. If we redraw the ring topology in a logical layout where the CPU is drawn as two "halves", we obtain the diagram in figure 6.4. When drawn this way, the ring looks like two separate logical linear chains. If we impose the condition that packets are not allowed to cross the cube from one chain to another (i.e., requests stay on either side of the red center line), then it is impossible to have a deadlock condition because all requests will flow from the CPU outward and all responses will flow in the opposite direction. That is, there will never be a case where one response goes to the left and another response goes to the right within the same chain. Imposing this condition obviates the need to keep separate reservation slots as discussed previously, but still allows the system to capture the latency and bandwidth benefits of a ring topology.



(a) Example of deadlock case

(b) One strategy to avoid deadlock

Figure 6.3: Deadlock in a ring topology

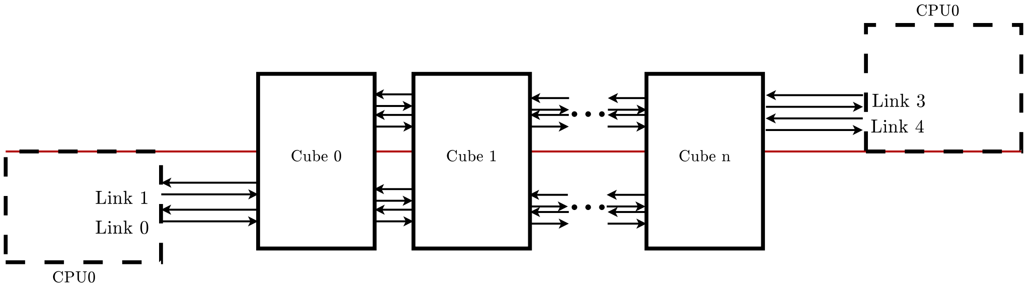


Figure 6.4: Logical representation of a ring topology. A ring is logically equivalent to two chains with the CPU at both ends. As long as requests are not allowed to cross the red line, no deadlock is possible.

The only drawback to this approach is that it reduces the number of available paths for routing. However, whether or not this restriction has any practical impact on performance is not immediately obvious and will be explored in section 6.5.

6.5 Cube Topology Random Stream Results

6.5.1 Route Heuristics

In this section we will discuss the impact of the topology using random stream simulations. The random stream simulation methodology for this section is the same as the one described in section 4.3 that was used for single cube simulations.

The topology problem is rather complex because of the number of variables that must be considered. In addition to the topology itself, we also consider the link choice heuristic, the route choice heuristic, the read/write ratio, and the number of cubes. Furthermore, sanity checking the results of the simulations is a difficult task due to the number of unique routes that are possible for certain topologies such as rings. In an attempt to visually show the flow of data through the network, the simulator tracks the route of every request along every hop through the simulator. The output file is processed by Graphviz to produce a graph diagram of each route segment and the utilization of each receiver-transmitter pair along that segment. Only actual data bytes are recorded since commands and packet overheads are simply a mechanism of requesting data movement and are not useful by themselves.

First, to get an overview of the result space, we consider only the 56% read/write ratio which corresponds to an optimal link throughput for 64 byte requests (see section 5.2). Figure 6.5 shows the cross product of the different link and route heuristics for the different topologies and numbers of cubes. Note that since we only implement the star topology for four cubes, we omit the results from this set of graphs.

The first feature of this graph is that the routing heuristics do not make much

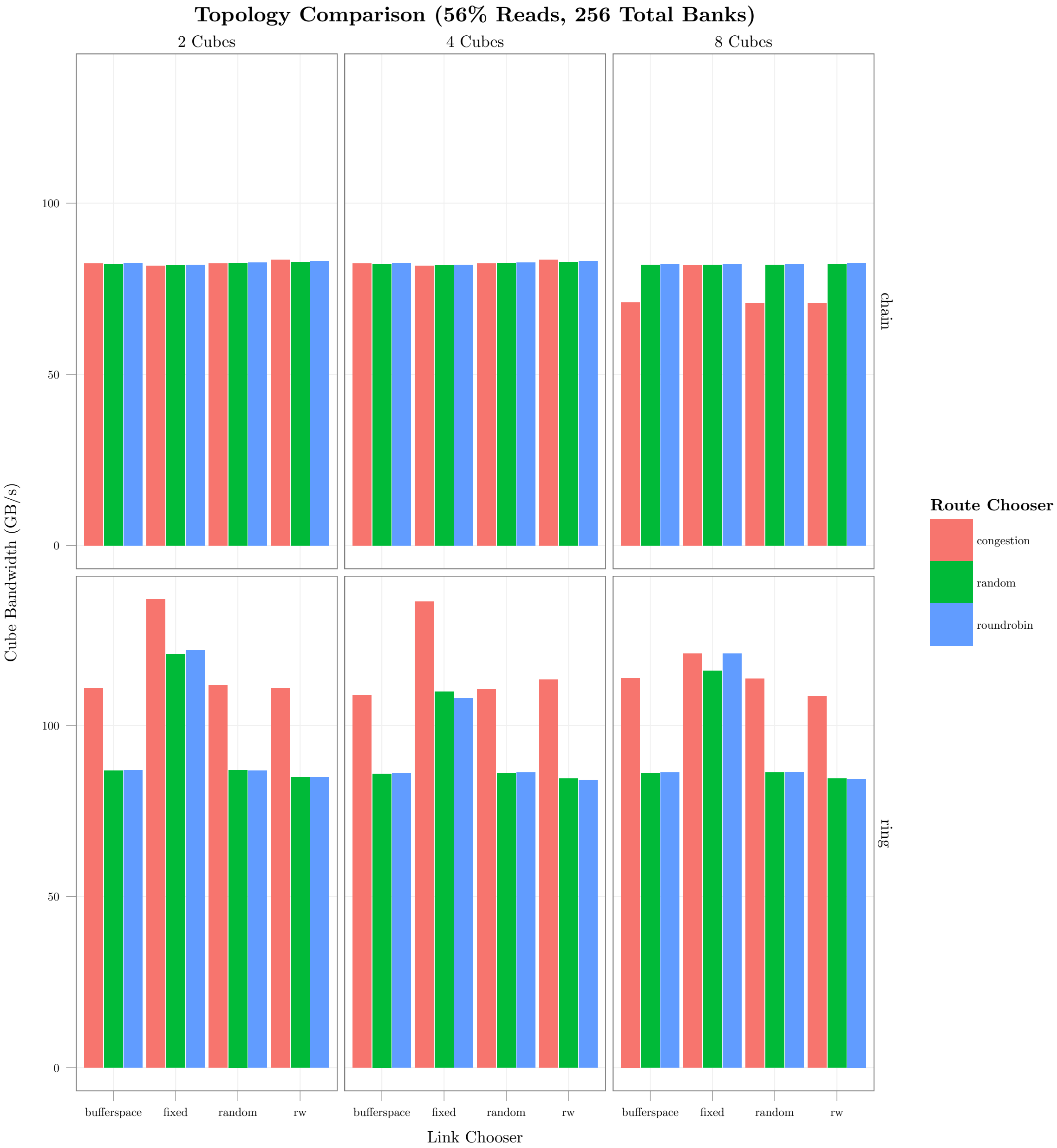


Figure 6.5: Bandwidth of various link and route heuristics with a 56% read/write ratio stream.

difference for a simple chain topology. This is expected since our simulated system only has two links connecting every CPU/cube pair; there are not very many unique paths to each vault. Therefore, the details of the route make only a small difference in the overall throughput of the system. We confirm this by examining the full set of read/write ratios for the chain in figure 6.6 and seeing that indeed, there is little change from any given heuristic.

One of the more surprising results from this data set is the fact that the "fixed" link chooser performs better than the other heuristics. The choice of link is made based on the bits that are directly above the transaction offset. That is, it shifts off the last 6 bits of the address for a 64 byte transaction and uses the two lowest bits to pick a link index. The default address mapping policy of the simulator places the vault bits in this same set of bits. Since the basic job of the routing logic is to assign paths between links and vaults, using the vault bits to assign a link index has the effect of separating traffic streams based on their destination vault. If we examine the ring configurations more closely over a range of read/write ratios (figure 6.7), the fixed scheme performs better than or nearly as well as the other schemes.

The chaining also goes against the previous result that the optimal read/write ratio for 64 byte requests is 56%. Figure 6.7 shows that memory streams with fewer reads and more writes are able to achieve better overall throughput in a ring topology. Because both write requests and return data may traverse the ring in either direction, these large packets interfere with the progress of both the read request and the read response. Since a full read return queue can stall the issuing of new read requests to the DRAM, the extra traffic in both directions can impede

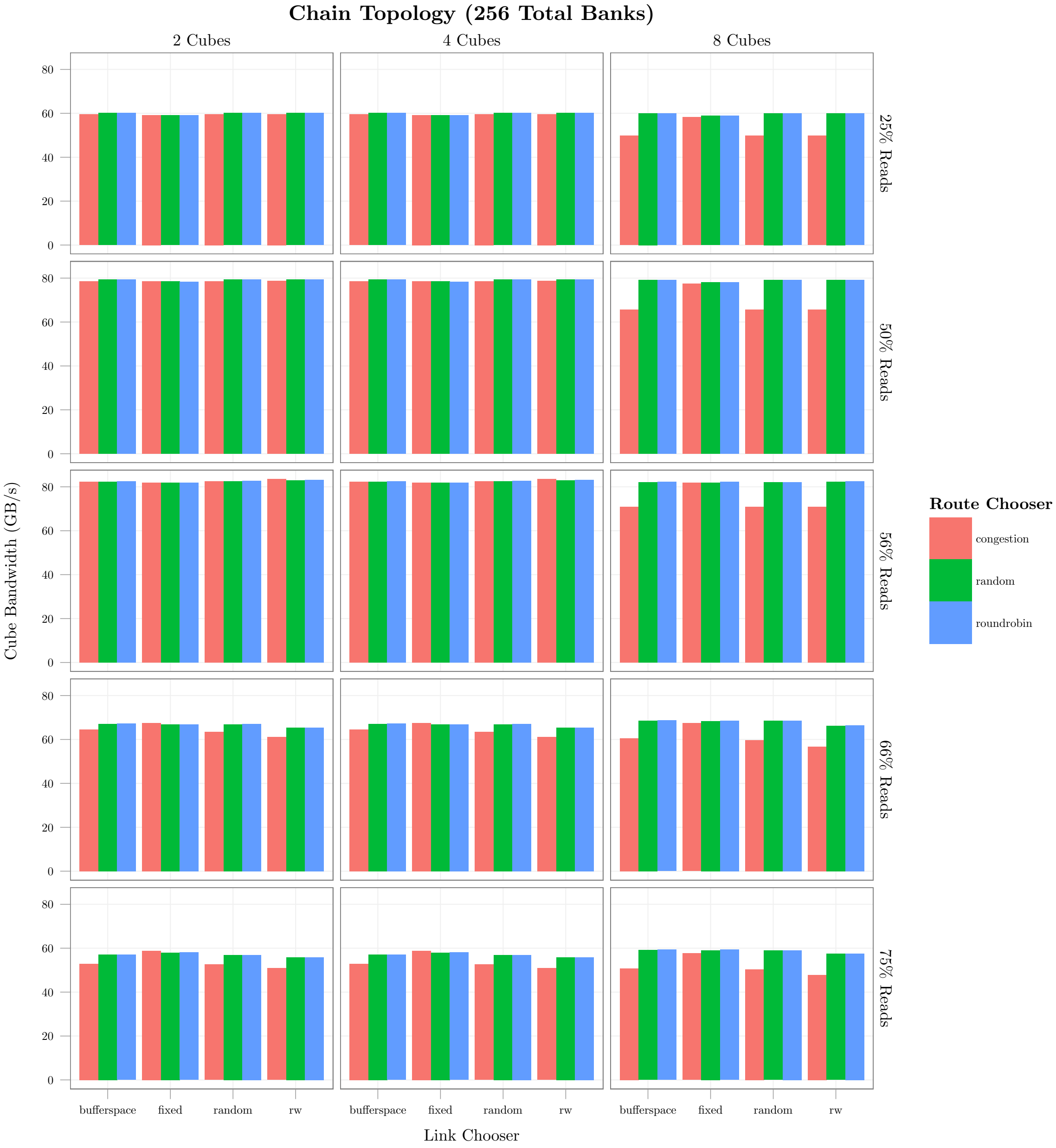


Figure 6.6: Bandwidth of various link and routing heuristics for a chain topology.

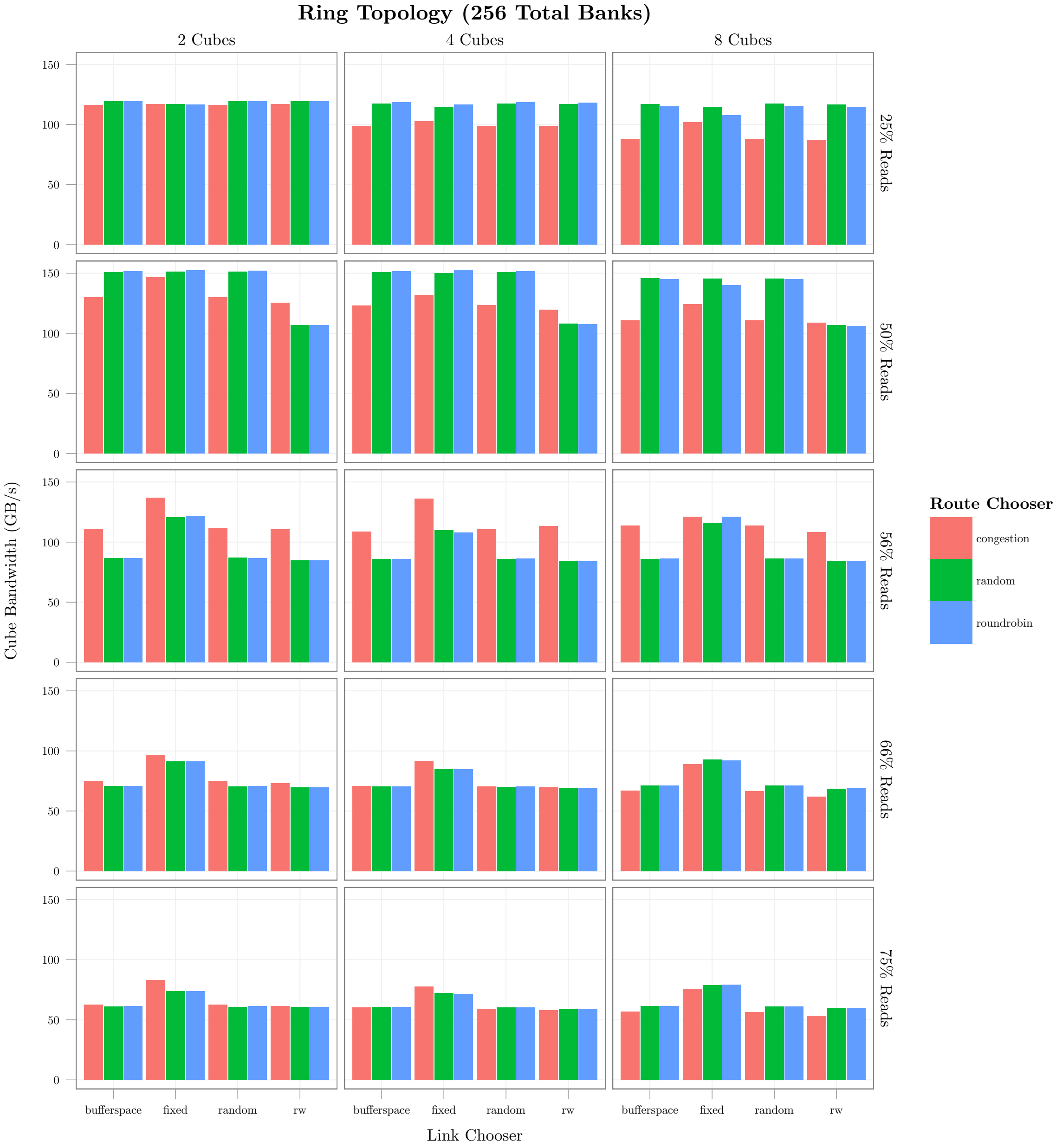


Figure 6.7: Bandwidth of various link and routing heuristics for a ring topology.

the progress of the DRAM. As writes can always be issued to the DRAM, they are not as sensitive to the passalong traffic in the cube and thus get better overall performance.

6.5.2 Full System Performance Impact

In this section, we illustrate the impact of HMC topologies the performance of the entire system. We configure a chain and a ring topology and vary the size of each to include one, two, or four cubes. We connect these different memory topologies to our MARSSx86 simulator instance and run several workloads.

Up until now, all of the simulations of multiple cubes have been only with random streams that are agnostic to address mapping. However, for a full-system experiment, we must choose a specific address mapping policy. In the initial run of this experiment, the cube bits were simply placed at the top of the address. This resulted in our simulations only sending requests to a single cube and thus failing to show any impact whatsoever. Therefore, we adjust the address mapping scheme so that the cube bits are further toward the lower bits of the address so that more spread is generated to each cube. We use a vault:bank:cube:rank:dram address mapping, which is a slight modification to the canonical closed page address mapping used earlier where cache lines most spread out (i.e., the vault and bank bits are in the lowest part of the address).

First, we summarize the number of requests arriving per epoch at each cube in a four cube topology with in figure 6.8. Each cube number is represented by a box plot with a different color with lines extending to the minimum and maximum. Indeed we can see that the requests are spread quite uniformly among the different cubes due to the address mapping scheme.

One of the key questions for evaluating the topology is how the latency varies for accesses to different cubes. This is especially important in a topology such as a chain where a request may have to traverse through the entire length of a chain of multiple cubes to reach its final destination. Figure 6.9 shows the latency of each cube over time for a four cube chain and ring. All of the workloads experience a fairly uniform latency difference when going to a cube farther down in a linear chain.

The right side of this figure that corresponding to the ring latencies is uneventful. The ring collapses all of the latency values down to a single line indicating that there is no significant latency penalty for requests going through passalong paths. As compared to the chain, the host has twice as many links to send requests and receive responses and also two cubes to send passalong traffic through instead of just one. Moreover, the number of hops to the farthest cube is halved as compared to a chain of equal size. Overall, the ring controls the latency distribution to different cubes within the topology rather well.

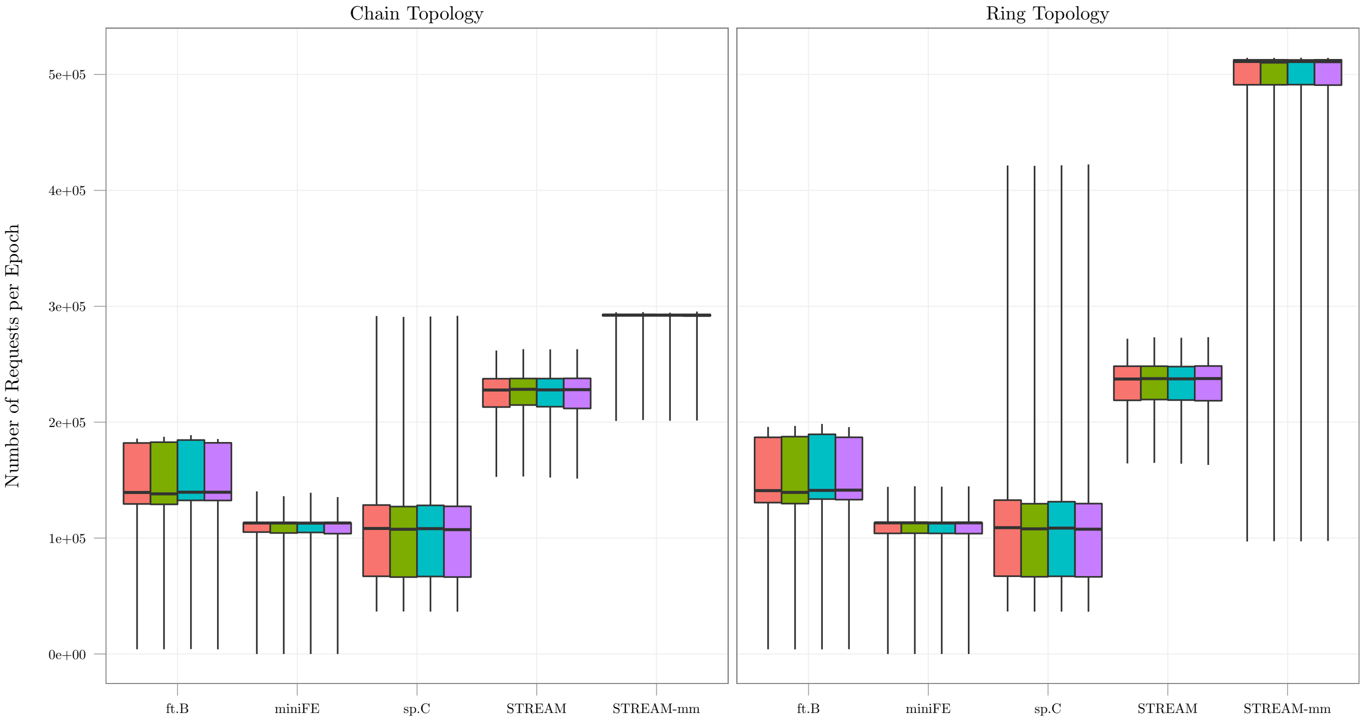


Figure 6.8: Box plot of average number of requests per epoch to each cube in four cube topologies. Requests are spread evenly among cubes since cube address bits are moved lower in the address.

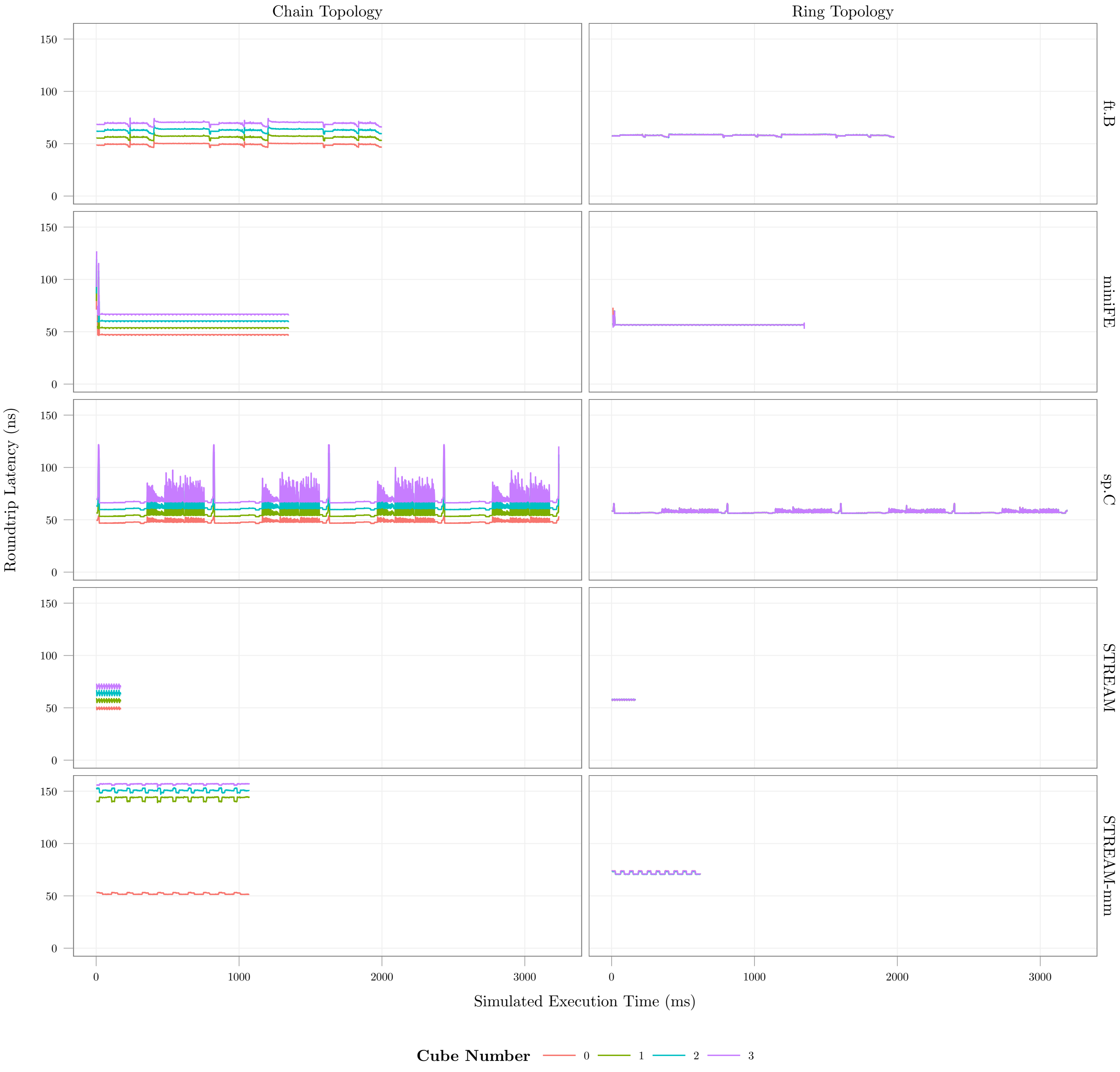


Figure 6.9: Round trip latency to different cubes in four cube topologies. Memory intensive applications experience a much higher latency when going to far away cubes in a chain. The ring topology is effective at controlling the request latency.

		Bandwidth (GB/s)		**Execution Time (ms)**		**Bandwidth Ratio**		**Speedup**	
	# of Cubes	chain	ring	chain	ring	chain	ring	chain	ring
ft.B									
	1	41.41	42.80	1929.69	1889.06	1.00	1.03	1.00	1.02
	2	41.73	42.61	1903.13	1865.63	1.01	1.03	1.01	1.03
	4	39.24	39.84	1998.44	1978.13	0.95	0.96	0.97	0.98
miniFE									
	1	20.56	20.65	1174.59	1169.11	1.00	1.00	1.00	1.00
	2	19.65	19.75	1228.46	1222.62	0.96	0.96	0.96	0.96
	4	17.92	17.90	1346.92	1348.44	0.87	0.87	0.87	0.87
sp.C									
	1	29.08	29.71	2932.81	2865.63	1.00	1.02	1.00	1.02
	2	28.38	28.94	3003.13	2946.88	0.98	1.00	0.98	1.00
	4	26.33	26.68	3237.50	3189.06	0.91	0.92	0.91	0.92
STREAM									
	1	55.64	60.30	161.26	148.77	1.00	1.08	1.00	1.08
	2	55.54	59.27	161.55	151.43	1.00	1.07	1.00	1.06
	4	51.43	53.23	174.50	168.62	0.92	0.96	0.92	0.96
STREAM-mm									
	1	65.11	118.20	1102.57	607.31	1.00	1.82	1.00	1.82
	2	67.02	118.93	1071.20	603.57	1.03	1.83	1.03	1.83
	4	67.00	115.37	1071.39	622.17	1.03	1.77	1.03	1.77

Table 6.1: Memory bandwidth and execution time impact of cube chaining. Bandwidth Ratio and Speedup are normalized to a single cube chain.

Finally, we examine the impact of cube chaining on overall memory throughput for different numbers of cubes. We graph the main memory bandwidth as seen from the CPU as a function of time for one, two, and four cube topologies in figure 6.10. We can see that as the latency results showed previously, there is a reasonable decrease in bandwidth when increasing the number of cubes in a linear chain. As the number of cubes increases, the ring is able to achieve higher bandwidth than the corresponding chain configuration. However, the impact on overall execution time is small, however, even for memory intensive workloads like STREAM.

These results show that there is an important trade-off between the capacity and performance of the HMC. As more cubes are added to a network of memory devices, there are more overheads associated with moving data around the network as compared to a single cube. However, as these results have shown, the trade-offs are quite reasonable in that adding capacity in the form of extra cubes only has a reasonably small impact on overall execution time. Furthermore, with operating system support that is aware of the non-uniformity of the address space, techniques could be applied to further hide the performance impacts associated with cube chaining.

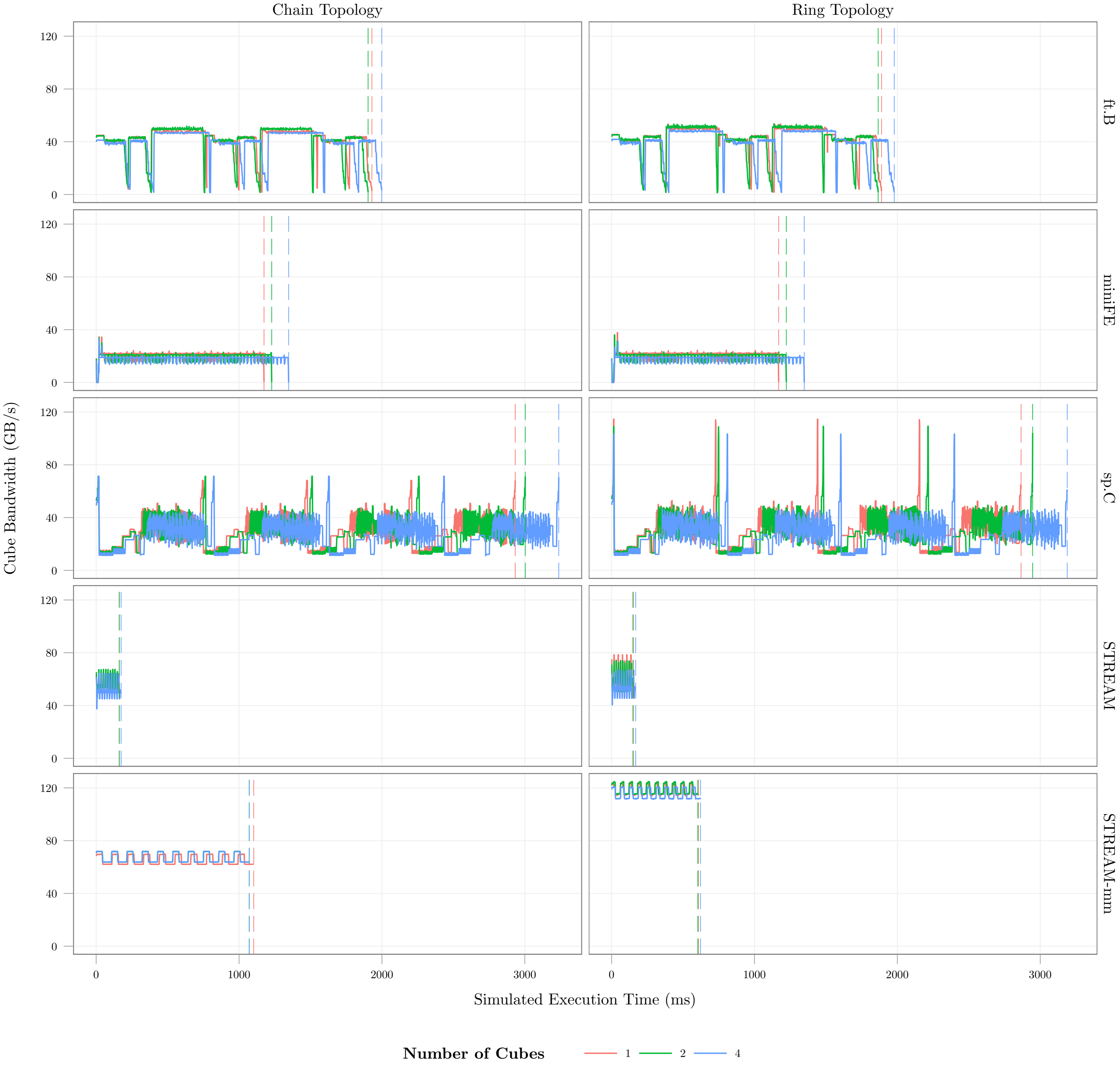


Figure 6.10: Memory bandwidth as seen from the CPU over time for topologies of varying size.

Chapter 7

Conclusion

The experiments in this thesis have shown that the Hybrid Memory Cube represents a paradigm shift in the structure and performance of the main memory system. Although the ideas of high speed signalling and 3D die stacking have been explored in the past, their combination inside of the memory system creates a device that can increase memory throughput by an order of magnitude over current technologies. This is a welcome change for the memory system, as researchers have been lamenting the "memory wall" for nearly two decades. The HMC provides a way to extend the life of DRAM as process technology scaling becomes more difficult.

We have demonstrated, however, that such a memory device must be configured carefully in order to make full use of the available memory-level parallelism and throughput. Link throughput must be over-provisioned with respect to TSV throughput due to the packet overheads and sensitivity to read/write ratio. Furthermore, we have touched upon the fact that such a memory system might expose CPU bottlenecks that were not previously problematic due to the reasonably slow memory system. That is, today's cache coherence protocols may become too much of a burden in the future when many cores attempt to access a high-throughput memory system. Experiments have shown that as the TSV process becomes more mature, adding more dies to the HMC stack can provide extra capacity without sacrificing performance. High levels of parallelism inside of the cube allow it to maintain high throughput even with a broad range of DRAM timings. Finally, we have shown that with the logic at the base of each memory stack and some careful consideration of topologies, the ability to connect cubes together topologies can also provide a path to increasing capacity while maintaining performance. It is likely that making the operating system kernel aware of the non-uniformity of the memory system can further increase the performance and lower the cost of cube chaining.

Overall, for certain memory intensive workloads, the HMC can decrease workload execution time as well as power consumption of the memory system while allowing vendors to innovate inside of the memory system.

Bibliography

[1] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, mar 1995.

[2] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 267–278, New York, NY, USA, 2009. ACM.

[3] Aniruddha N. Udipi, Naveen Muralimanohar, Rajeev Balasubramonian, Al Davis, and Norman P. Jouppi. Combining memory and a controller with photonics through 3D-stacking to enable scalable and energy-efficient systems. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 425–436, New York, NY, USA, 2011. ACM.

[4] Micron. 4Gb: x4, x8, x16 DDR3 SDRAM Features. `http://www.micron.com/~/media/Documents/Products/Data%20Sheet/DRAM/DDR3/4Gb_DDR3_SDRAM.pdf`, 2009.

[5] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Pub, 2007.

[6] Jung-Ho Ahn, J. Leverich, R.S. Schreiber, and N.P. Jouppi. Multicore dimm: an energy efficient memory module with independently controlled drams. *Computer Architecture Letters*, 8(1):5–8, 2009.

[7] JEDEC. Main Memory: DDR3 & DDR4 SDRAM. `http://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4-sdram`.

[8] Todd Legler. Choosing the DRAM with Complex System Considerations. In *Embedded Systems Conference*, 2012.

[9] D. Wang. Importance of migrating to DDR4.

[10] Graham Allan. Ddr4 bank groups in embedded applications. `http://www.synopsys.com/Company/Publications/DWTB/Pages/dwtb-ddr4-bank-groups-2013Q2.aspx`, 2013.

[11] Joel Hruska. Haswell-E to offer DDR4 support, up to eight cores in 2014. `http://www.extremetech.com/computing/158824-haswell-e-to-offer-ddr4-support-up-to-eight-cores-in-2014`, 2013.

[12] Richard Swinburne. DDR4: What we can Expect. `http://www.bit-tech.net/hardware/memory/2010/08/26/ddr4-what-we-can-expect/2`, August 2010.

[13] Samsung. Samsung DDR4 SDRAM. http://www.samsung.com/global/business/semiconductor/file/media/DDR4_Brochure-0.pdf, 2013.

[14] Samsung. What is a Load Reduced Dual-inline Memory Module (LRDIMM)?

[15] Inphi. Introducing LRDIMM – A New Class of Memory Modules. www.inphi.com/products/whitepapers/Inphi_LRDIMM_whitepaper_Final.pdf, 2011.

[16] S. Kuppahalli. Inphi and Samsung Demonstrated LRDIMM Technology at IBM EDGE 2013.

[17] B. Ganesh, A. Jaleel, D. Wang, and B. Jacob. Fully-buffered DIMM memory architectures: Understanding mechanisms, overheads and scaling. pages 109–120, 2007.

[18] Micron. TN-47-21: FBDIMM – Channel Utilization (Bandwidth and Power) Scalable Power. Technical report, 2006.

[19] Elliott Cooper-Balis, Paul Rosenfeld, and Bruce Jacob. Buffer-on-board memory systems. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 392–403, june 2012.

[20] Joel Hruska. Samsung pushes ahead with 20nm DDR3 RAM, signaling uncertainty about DDR4, March 2014.

[21] J. Thomas Pawlowski. Hybrid Memory Cube (HMC). In *Hot Chips 23*, August 2011.

[22] Hybrid Memory Cube Specification 1.0. Technical report, Hybrid Memory Cube Consortium, 2013.

[23] Joe Jeddeloh. PCI-e FLASH and Stacked DRAM for Scalable Systems, May 2013.

[24] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88, 2012.

[25] M. Facchini, T. Carlson, A. Vignon, M. Palkovic, F. Catthoor, W. Dehaene, L. Benini, and P. Marchal. System-level power/performance evaluation of 3D stacked DRAMs for mobile applications. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 923–928, april 2009.

[26] Justin Rattner. Intel Speaks about the Hybrid Memory Cube. http://www.youtube.com/watch?v=VMHgu_MKjkQ, 2011.

[27] Hybrid Memory Cube: Experimental DRAM. In *Intel Developer Forum*, 2011.

[28] Micron. Revolutionary Advancements in Memory Performance . http://www.youtube.com/watch?v=kaV2nZSkw8A.

[29] K.T. Malladi, F.A. Nothaft, K. Periyathambi, B.C. Lee, C. Kozyrakis, and M. Horowitz. Towards energy-proportional datacenter memory with mobile dram. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 37–48, 2012.

[30] C. Weis, N. Wehn, L. Igor, and L. Benini. Design space exploration for 3D-stacked DRAMs. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, march 2011.

[31] Dana Vantrease, Robert Schreiber, Matteo Monchiero, Moray McLaren, Norman P. Jouppi, Marco Fiorentino, Al Davis, Nathan Binkert, Raymond G. Beausoleil, and Jung Ho Ahn. Corona: System Implications of Emerging Nanophotonic Technology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 153–164, Washington, DC, USA, 2008. IEEE Computer Society.

[32] Gurtej Sandhu. DRAM Scaling & Bandwidth Challenges. In *NSF Workshop on Emerging Technologies for Interconnects (WETI)*, 2012.

[33] M. B. Kleiner, S. A. Kuhn, P. Ramm, and W. Weber. Performance improvement of the memory hierarchy of RISC-systems by application of 3-D technology. *Components, Packaging, and Manufacturing Technology, Part B: Advanced Packaging, IEEE Transactions on*, 19(4):709–718, nov 1996.

[34] C. C. Liu, I. Ganusov, M. Burtscher, and Sandip Tiwari. Bridging the processor-memory performance gap with 3D IC technology. *Design Test of Computers, IEEE*, 22(6):556–564, nov.-dec. 2005.

[35] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. Die Stacking (3D) Microarchitecture. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 469–479, dec. 2006.

[36] Taeho Kgil, Shaun D'Souza, Ali Saidi, Nathan Binkert, Ronald Dreslinski, Trevor Mudge, Steven Reinhardt, and Krisztian Flautner. PicoServer: using 3D stacking technology to enable a compact energy efficient chip multiprocessor. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 117–128, New York, NY, USA, 2006. ACM.

[37] Gian Luca Loi, Banit Agrawal, Navin Srivastava, Sheng-Chih Lin, Timothy Sherwood, and Kaustav Banerjee. A thermally-aware performance analysis of vertically integrated (3-D) processor-memory hierarchy. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 991–996, New York, NY, USA, 2006. ACM.

[38] Gabriel H. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 453–464, Washington, DC, USA, 2008. IEEE Computer Society.

[39] Ke Chen, Sheng Li, N. Muralimanohar, Jung Ho Ahn, J. B. Brockman, and N. P. Jouppi. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 33–38, march 2012.

[40] Aniruddha N. Udipi, Naveen Muralimanohar, Niladrish Chatterjee, Rajeev Balasubramonian, Al Davis, and Norman P. Jouppi. Rethinking DRAM design and organization for energy-constrained multi-cores. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 175–186, New York, NY, USA, 2010. ACM.

[41] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. Memory-centric System Interconnect Design with Hybrid Memory Cubes. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 145–156, Piscataway, NJ, USA, 2013. IEEE Press.

[42] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, and Vijayalakshmi Srinivasan. NDC: Analyzing the Impact of 3D-Stacked Memory+ Logic Devices on MapReduce Workloads. In *ISPASS*, 2014.

[43] Thomas Kinsley and Aron Lunde. Inside the Hybrid Memory Cube, September 2013.

[44] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. pages 128–138, 2000.

[45] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

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

[47] A. Patel and F. Afram. MARSSx86: Micro-ARchitectural and System Simulator for x86-based Systems, 2010.

[48] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16 –19, jan.-june 2011.

[49] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, dec 1995.

[50] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[51] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[52] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 2009.

[53] Wang. *Modern DRAM Memory Systems: Performance Analysis and a High Performance, Power-Constrained DRAM Scheduling Algorithm.* PhD thesis, University of Maryland, 2005.