INAUGURAL - DISSERTATION

zur

Erlangung der Doktorwürde

der

Naturwissenschaftlich-Mathematischen Gesamtfakultät

der

Ruprecht – Karls – Universität

Heidelberg

vorgelegt von

Myles Glen Watson

aus: Twin Falls, Idaho, Vereinigten Staaten von Amerika

Tag der mündlichen Prüfung:

Applications for Packetized Memory Interfaces

Betreuer: Professor Dr. Ulrich Brüning

Abstract

The performance of the memory subsystem has a large impact on the performance of modern computer systems. Many important applications are memory bound and others are expected to become memory bound in the future. The importance of memory performance makes it imperative to understand and optimize the interactions between applications and the system architecture. Prototyping and exploring various configurations of memory systems can give important insights, but current memory interfaces are limited in the amount of flexibility they provide. This inflexibility stems primarily from the fixed timing of the memory interface. Packetized memory interfaces abstract away the underlying timing characteristics of the memory technology and allow greater flexibility in the design of memory hierarchies. This work uses packetized interfaces to explore memory hierarchy designs and prototype a novel network attached memory. Since current processors do not support packetized memory interfaces, a coherent processor bus is used as a memory interface for the DiskRAM project. The Hybrid Memory Cube (HMC) packetized memory interface is also presented and used to prototype network-attached memory. The HMC interface is discussed in detail, along with the design and implementation of a Universal Verification Component (UVC) environment. The convergence of network and memory interfaces is also predicted.

Zusammenfassung

Die Leistung des Speichersystems hat einen großen Einfluss auf die Leistungsfähigkeit moderner Computersysteme. Viele wichtige Anwendungsprogramme sind von der Speichergeschwindigkeit limitiert und es wird erwartet, dass in Zukunft dieses Problem weiter zunimmt. Der Einfluss der Speicherleistung macht es zwingend erforderlich, die Interaktionen zwischen Anwendungen und der Systemarchitektur zu verstehen und zu optimieren. Prototypen und das Evaluieren verschiedener Konfigurationen von Speichersysteme können wichtige Einblicke geben. Die aktuellen Speicherschnittstellen bieten allerdings nur eingeschränkte Flexibilität. Diese mangelnde Flexibilität resultiert im Wesentlichen aus dem festen Timing der Speicherschnittstelle. Botschaftenorientierte Speicherschnittstellen abstrahieren die unterliegende Zeiteigenschaften der Speichertechnologie und ermöglichen eine größere Flexibilität bei der Gestaltung von Speicherhierarchien. Diese Dissertation verwendet botschaftenorientierte Schnittstellen um den Entwurf von Speicherhierarchien zu erforschen und um einen Prototyp eines neuartigen Speichers zu bauen, der nur über eine Netzwerkschittstelle angebunden wird. Diese Speichertechnologie wird als Network Attached Memory (NAM) bezeichnet. Aktuelle Prozessoren verfügen leider nicht über solche boschaftenorientierte Speicherschnittstellen, deshalb wird im ersten Projekt ein kohärentes Prozessorinterface als Speicherschnittstelle für DiskRAM verwendet. Weiterhin wird das neuartige Hybrid-Memory Cube (HMC) Interface vorgestellt und verwendet, um das NAM zu modelieren. Die HMC-Schnittstelle wird im Detail diskutiert, um die Vorteile des Interfaces zu zeigen. In der Arbeit wurde die Konzeption und Umsetzung einer Universal Verification Component (UVC) für die HMC-Schnittsctelle entwickelt. Die zukunftige Zusammenführung von Netzwerk- und Speicherschnittstellen wird vorhergesagt.

For Christie

Acknowledgements

I want to thank those who believed in me, supported me, and encouraged me to continue.

Contents

List of Figures

List of Tables

Acronyms

CERAM	Correlated Electron RAM.
CRC	Cyclic Redundancy Code.
DDR	Double Data Rate.
DEEP	Dynamical Exascale Entry Platform.
DEEP-ER	DEEP-Extended Reach.
DRAM	Dynamic RAM.
DUT	Device Under Test.
ECC	Error Correcting Code.
EEPROM	Electrically-Erasable Programmable Read-Only Memory.
EPROM	Electrically Programmable Read-Only Memory.
FPGA	Field Programmable Gate Array.
FRAM	Ferroelectric RAM.
HMC	Hybrid Memory Cube.
MB/s	MegaBytes per second.
MLC	Multi-Level storage Cell.
MRAM	Magnetoresistive RAM.
MT/s	MegaTransfers per second.
MTJ	Magnetic Tunneling Junction.
NAM	Network Attached Memory.
NVRAM	Non-Volatile RAM.

PCI	Peripheral Components Interconnect.
PCIe	PCI Express.
PCM	Phase-Change Memory.
RAM	Random-Access Memory.
RRAM	Resistive RAM.
SATA	Serial ATA.
SRAM	Static RAM.
STT-RAM	Spin-Torque-Transfer RAM.
UVM	Universal Verification Methodology.

1 Introduction

The basic architecture of a computer system, shown in Figure 1.1, remained largely unchanged for many years. Technology scaling provided vast increases in the amount of resources available to each component of the system, and huge performance gains were made. The pace of technology scaling has now changed, which affects system design. In order to conserve power, specialized processors are being designed to target various segments. The effects of these changes to the processor architecture on the memory hierarchy are not trivial, and are important for optimizing the performance of future systems. The goal of this work is to motivate the adoption of packetized memory interfaces. This would provide more flexibility to designers and facilitate the creation of prototype systems.

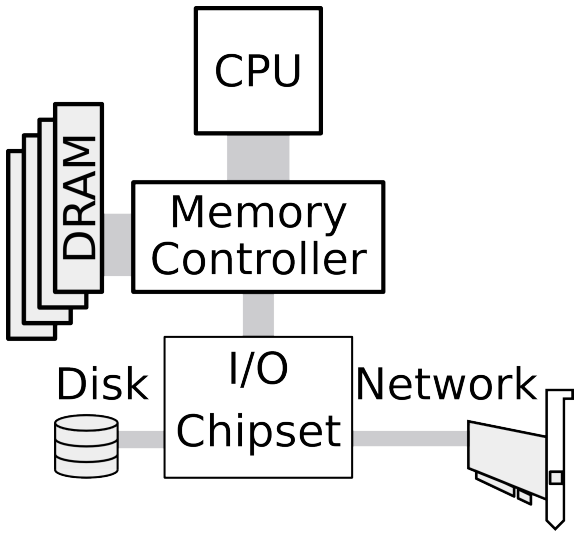

Figure 1.1: The block diagram of a computer system.

This chapter briefly reviews Moore's law and Dennard scaling, discusses its effect on the components of computer systems, and posits that communication between the components of the computer system must be optimized. It then calls for flexible and extensible interfaces, which allow systems to be prototyped and built with different types and amounts of storage, processing, and networking capabilities. It stresses the importance of packetized memory interfaces in the realization of this goal, and

outlines the topics and contributions of this work.

1.1 Technology Scaling

The exponential scaling of silicon fabrication technology has been an engine for technological progress for many decades. This scaling can be divided into two related parts, Moore's Law and Dennard scaling.

Moore's law was formulated by Gordon Moore in 1965 and refined in 1975 [1]. It predicts the rate of improvement of integration for the semiconductor industry. Generally, it states that the number of components which can be combined economically in a single integrated circuit would double every two years. It has proven to accurately predict the increase in complexity for integrated circuits for many years.

Dennard scaling was described by Robert Dennard in 1974 [2] and relates transistor scaling to threshold voltage, frequency, power density, power dissipation, and doping concentrations. Frequency, threshold voltage, and power have not continued to scale, although Moore's Law has continued to predict the density of transistors. The end of Dennard scaling has shifted the focus of processor manufacturers from maximizing frequency to increasing the number of processing cores per chip.

1.2 Effects on Processors and Memory

Two of the most important components of a computer system are the processor and the memory. Moore's law and Dennard scaling caused these components to be optimized differently. This section describes how they have been optimized, and why.

As technology scaled, processors became larger and more complex, integrating many functions that were performed by separate integrated circuits. This integration also resulted in standardization, in part because the cost of adding additional functionality to an existing circuit is low compared to manufacturing a separate integrated circuit. Since the operating frequency of processors was also increasing, caches were added to reduce the average latency to memory. The size of these caches increased, and multiple levels of caches were added to on-chip memory hierarchies. This continued until the majority of the die area was dedicated to the memory hierarchy [3]. This is due in part to the fact that the latency for retrieving information from the memory, or for any off-chip transfer, was so high compared to the latency of performing calculations. The end of Dennard scaling shifted the focus of designers to adding

multiple processors per chip.

Technology scaling affected computer memory chips differently. These chips were optimized primarily for density, then for transfer bandwidth. The access latency of RAM decreased, but much more slowly. Capacity and price are more important than access time, in part because of the deep memory hierarchies incorporated in processors. The end of Dennard scaling for memory increased the number of banks per chip and is forcing a "rethinking" of memory chip design [4].

1.3 Optimizing Communication

One of the principles used to guide the optimization of computer systems is the continuously increasing cost of communication relative to computation, as measured in terms of power and latency. Figure 1.2 illustrates the important communication interfaces in computer systems. This work focuses on understanding and improving the interface between the processor and memory.

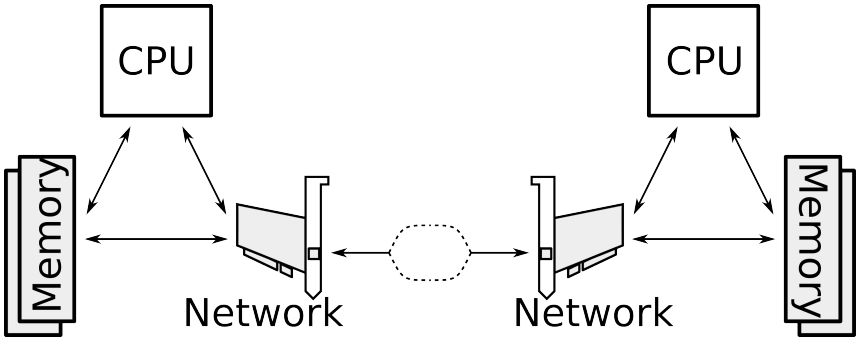

Figure 1.2: Communication paths that need to be optimized to improve the performance of future computer systems.

1.4 Vision

An ideal memory interface would be flexible and extensible, so that multiple memory technologies can be connected to it. It would also provide a wider range of commands than just reads and writes, allowing some processing to be performed as close to the data as possible. In order to provide this flexibility, it would have to be a packetized interface.

1.4.1 Packetized Memory Interfaces

Packetized interfaces provide flexibility because they separate the pins of the interface from the data that is transferred. They can thus be scaled independently with respect to frequency and interface width. They can also be more easily extended, since adding address bits or command fields can be accomplished with changes in the packet definition, without the addition of new pins. Since the timing of responses is not fixed, they can be used to address multiple storage technologies. This flexibility comes at the price of some extra latency and packet overhead, but enables novel system architectures and makes prototyping much easier.

1.5 Contributions

This work demonstrates the utility of packetized interfaces for prototyping with memory interfaces. It motivates the need for flexible memory interfaces by describing some of the competing memory technologies, presents two applications of packetized interfaces to memory interfaces, and gives an in-depth look at a commercial packetized memory interface.

The first application, DiskRAM, connects a memory controller prototyped with an Field Programmable Gate Array (FPGA) to a commercial processor. Using the packetized interface allows the memory controller to service memory requests. This would not be possible with the fixed timing of a memory interface.

The second application, Network Attached Memory (NAM), employs a network interface to add globally addressable storage in a distributed machine. This storage is designed to be used to enhance the performance and reliability of I/O and checkpointing operations. Since the network interface is packetized, customized delays and global operations can also be added to the implementation. This makes it an attractive research vehicle for exploring the use of future memory technologies and the acceleration of collective operations.

This work also analyzes of the implementation of the Hybrid Memory Cube (HMC) interface. The HMC interface is a commercial packetized memory interface, which is designed to address 3D stacks of DRAM. The protocol is discussed in detail, along with insights gained from the implementation of a Universal Verification Methodology (UVM) environment for testing an HMC controller.

1.6 Outline

Chapter 2 presents an overview of solid-state memory technologies and their attributes. This discussion highlights the uncertainty surrounding the future of solid-state storage, as there are many promising technologies with no clear winner. Chapter 3 describes DiskRAM, a project which used the processor interface as a packetized memory interface. This view of the interface enabled the implementation of demand paging in hardware. Chapter 4 describes Network-Attached Memory, which is part of the European Union Dynamical Exascale Entry Platform - Extended Reach (DEEP-ER) project. This project uses the network interface as a packetized memory interface, in order to prototype the addition of globally addressable storage in the network. The controller is designed to include programmable delays to emulate various types of solid-state storage. Chapter 5 describes the Hybrid Memory Cube (HMC) interface, which was used for implementing the Network-Attached Memory described in Chapter 4. It also describes the implementation of a Universal Verification Methodology (UVM) environment for testing the HMC. It concludes by presenting a design for scalable packet processing to decrease the prototyping effort and a discussion of the possible future convergence of memory and network interfaces. Chapter 6 presents the conclusion and suggests future research directions.

2 Solid-State Storage Technology Overview

Storage needs have increased with the increasing speed of computation. For much of the history of solid-state computation, Static RAM (SRAM) has dominated on-chip storage, and Dynamic RAM (DRAM) has been the designer's choice for main memory. As fabrication processes continue to shrink and the power budgets for large machines are taken into account, it is unclear if SRAM and DRAM will maintain their historical positions in the memory hierarchy. This chapter introduces some domain-specific metrics for storage, then presents some of the candidate technologies for storage in future supercomputers. It ends with a description of techniques for increasing density and the recommendation to use packetized memory interfaces.

2.1 Metrics

Many metrics can be used to compare memory technologies. Some of the most important metrics are volatility and persistence, density, price, capacity, latency, bandwidth, and energy.

2.1.1 Volatility and Persistence

In this work, storage is considered to be volatile if the device could lose data when power is removed. Volatility can be an important consideration when designing systems for reliability since power supplies can fail, and many other types of failures can only be corrected by turning off the power supply. Volatility implies that there is an active component to the power consumption of a device, even when it is not being accessed. In large systems, this can influence scaling. Persistence is the opposite of volatility.

2.1.2 Density

Density refers to the number of bits of storage per unit area. For most technologies, this is inversely correlated to price. Density can be expressed as the average area needed to store a single bit, in terms of the minimum feature size, F.

2.1.3 Price

Price is an important metric for memory technologies, but is seldom used alone. Price is frequently used in conjunction with metrics such as capacity and bandwidth. Using these derived metrics enables the price comparison of vastly different technologies applied to a specific function.

2.1.4 Capacity

Capacity is a measure of the total number of bits that can be stored in a device. For solid-state applications, the device is most often a chip or a package.

2.1.5 Latency

Latency refers to the time elapsed between the initiation of a command and its completion. For commands which return a response, such as a read, this is easily measured. For commands without a response, the latency can be defined as the time until another command can be initiated. Latencies can vary greatly across technologies and for different operations.

2.1.6 Bandwidth

Bandwidth refers to the transfer rate of an interface. With memory technologies, it usually refers to the maximum theoretical data transfer bandwidth. This is defined as the number of data bits (or bytes) transferred per time interval, divided by the length of the time interval. The JEDEC DDR3 specification [5] refers to DDR3-800 x4 chips meaning 800 MegaTransfers per second (MT/s) with a data width of 4 bits. This chip has a maximum bandwidth of 400 MegaBytes per second (MB/s) since there are 8 bits per byte. A DDR3-1600 x8 chip, with 1600 MT/s has a maximum bandwidth of 1600 MB/s per chip. Some confusion may arise because DDR3 chips are often integrated into 64-bit modules, which are named according to their bandwidth. For example, PC3-12800 modules have a maximum bandwidth of 12800 MB/s.

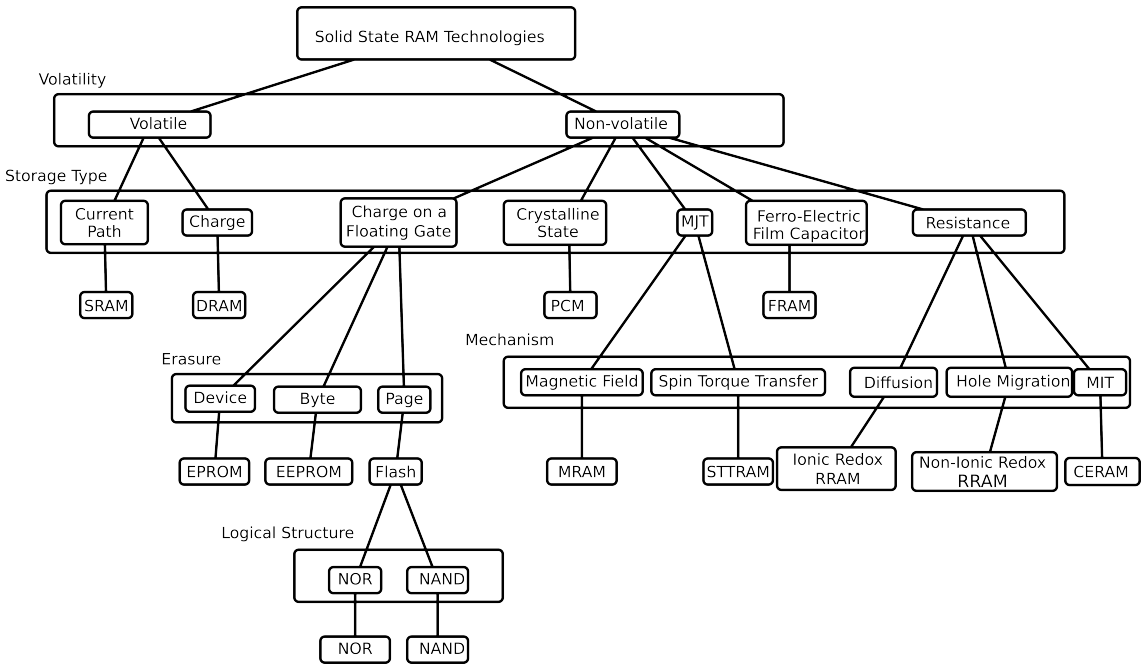

Figure 2.1: A design space diagram for solid-state storage.

2.1.7 Energy

Energy becomes an important consideration as capacity scales. Energy is normally reported according to its purpose. The energy required to read and write a single bit varies dramatically among storage technologies. In addition, some technologies require energy to maintain the storage state, some require periodic refreshes, and some require energy to prepare a region of storage for writing.

2.1.8 Metrics Summary

The existence of so many metrics generates a complex design space for exploration. In this design space there are many types of Random-Access Memory (RAM), which may be used for various purposes in the design of computing systems. Computers approaching exascale may make use of many types of RAM as part of specialized subsystems and components.

The following discussion of RAM technologies starts with volatile RAM technologies and then presents Non-Volatile RAM (NVRAM) technologies. Figure 2.1 shows the design space to be covered. Named boxes represent axes in the design space.

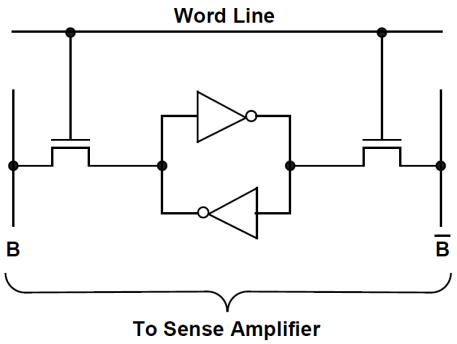

Figure 2.2: The circuit diagram of an SRAM cell with cross-coupled inverters. [6]

2.2 Volatile RAM

The two most important types of volatile RAM used in computer systems are SRAM and DRAM. Both technologies provide fast read and write times, which are nearly symmetrical. Their widespread use has meant that more effort has been invested in their design and optimization than for those of other technologies.

2.2.1 Static RAM

A classical SRAM cell consists of a pair of cross-coupled inverters, as shown in Figure 2.2. The logic state is stored as a voltage level between their outputs and respective inputs. When the word line W is driven, the access transistors connect the bit lines B and $\bar{B}$ to the sense amplifiers so that the value can be read. During a write operation, a new value is driven onto the bit lines. The lines are driven with enough current to overcome the driving strength of the inverters, and the state of the logic cell changes to the new value.

The basic SRAM cell is the six-transistor (6T) SRAM cell, which is diagrammed in Figure 2.3. Four of the transistors implement the cross-coupled inverters from Figure 2.2. The other two transistors are the access transistors, which connect the cell to the bit lines when selected by the word line. Many chips utilize a four-transistor (4T) design, which approximates 6T SRAM by replacing two of the transistors in the inverters with resistors, as shown in Figure 2.4. The 4T SRAM cell is superior to the 6T cell in terms of density, at the cost of higher power consumption and slower switching times [6].

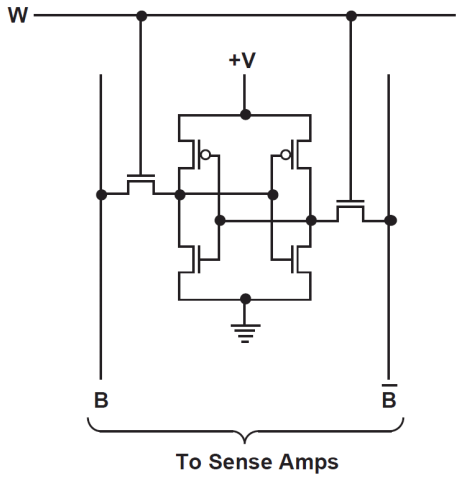

Figure 2.3: The circuit diagram for a six transistor (6T) SRAM cell [6].

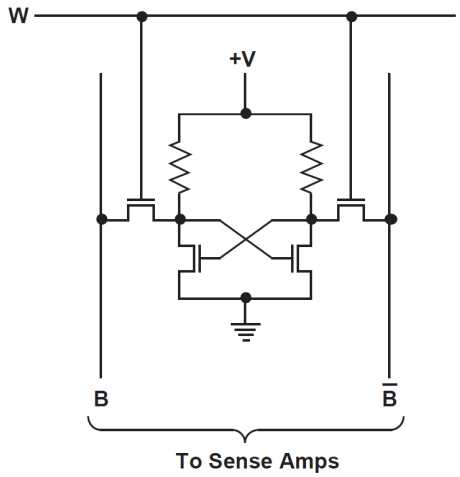

Figure 2.4: The circuit diagram for a four transistor (4T) SRAM cell [6].

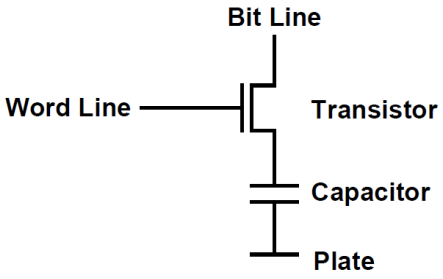

Figure 2.5: A DRAM cell with one transistor and one capacitor (1T1C) [7].

2.2.2 Dynamic RAM

DRAM cells consist of an access transistor and a capacitor (1T+1C), as shown in Figure 2.5. The logic state is stored as a charge on the capacitor, which can be accessed when the transistor is activated. Using a capacitor to store data has important consequences in terms of volatility, density, organization, and power consumption.

It is important to realize that reading is a destructive operation for a DRAM cell, since it releases the charge stored on the capacitor. In order for a stored bit to be correctly interpreted, the capacitor needs to have enough stored charge to raise the voltage of the bit line above the input threshold of the sense amplifiers. The charge that can be stored on a capacitor is directly proportional to voltage and capacitance. In order to reduce the power consumption of DRAM devices, the voltage has been

reduced in successive generations. To store enough charge at lower voltages, the capacitance must be increased. Since capacitance is directly proportional to the area of the capacitor, much effort has been invested in finding ways to increase the effective area of the capacitor without increasing the amount of silicon area required for each bit cell [7].

Ideally, the capacitors would hold their charge indefinitely. Real capacitors leak, or lose charge over time. In a DRAM cell, the access transistor allows some current to flow even when it is turned off. This leakage current is proportional to the voltage across the transistor. In order to reduce leakage, the "plate" in Figure 2.5 is held at a voltage between ground and the write voltage when not writing or reading. Both types of leakage contribute to the loss of stored data. This means that the state of the capacitor must be read before it is lost and then re-written to maintain a large enough charge on the capacitor to avoid data loss. Modern DRAM devices need to be refreshed on the order of every few milliseconds. After each read, the data must be written back. The read circuitry can thus be used for refresh.

DRAM devices are organized into rows and columns, as shown in 2.6. In order to read stored data, the bit lines are precharged to the write voltage divided by two. When the correct row is selected, the stored values are driven onto the bit lines, converted into logical values by the sense amplifier, and stored in the row buffer. At this point, the column is selected, and the stored value from the row buffer is driven onto the data pins. Subsequent read operations that target the same row can occur much faster, because the value is already stored in the row buffer. This can result in significantly reduced latency and power consumption when the locality of the workload is high. The data that is loaded into the row buffer is also referred to as an open page.

2.3 Non-Volatile RAM

Data stored in NVRAM is retained without consuming any power. This section gives an overview of various NVRAM technologies and their advantages and disadvantages. Flash memory is discussed first because of its large market share, then Phase-Change Memory (PCM), Magnetoresistive RAM (MRAM), Spin-Torque-Transfer RAM (STT-RAM), and Resistive RAM (RRAM) are presented.

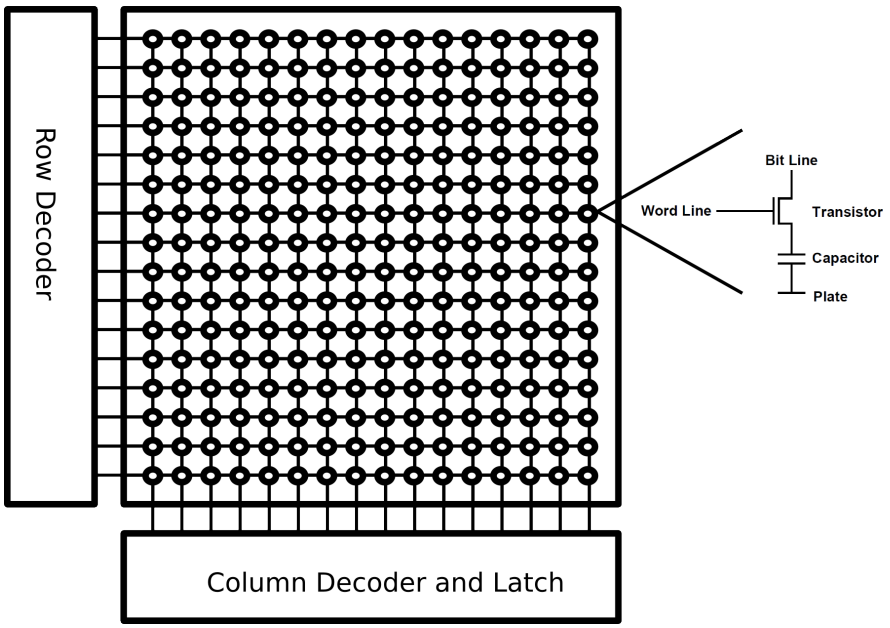

Figure 2.6: DRAM rows and columns. The DRAM in this figure consists of 16 rows and 16 columns with a storage cell at the intersections of the bit and word lines.

2.3.1 NOR and NAND Flash

Flash memory is currently the dominant non-volatile memory technology. It is a type of Electrically-Erasable Programmable Read-Only Memory (EEPROM) which is organized into pages to increase the speed of erase (and therefore write) operations. Understanding how Electrically Programmable Read-Only Memory (EPROM) and EEPROM work makes it easier to understand flash memory. EPROM, EEPROM, and flash memory must be erased before they are written. Their differences lie in how they are read, written, and erased.

EPROM and EEPROM are mature NVRAM technologies which store information in the form of electrons trapped on the "floating gate" of a transistor. Figure 2.7 shows a cross section of an EPROM transistor with a floating gate. The floating gate is electrically isolated from the gate by a dialectric layer, forming a capacitor. When electrons are trapped on the floating gate, they affect the conductivity of the channel of the transistor. This change in conductivity can be used to read the state of the stored bit. In order to remove the trapped electrons in an EPROM, or erase the stored data, the device is exposed to ultraviolet light. This allows the trapped electrons to move through the insulating oxide and resets all bits to their base state. EPROM can not be partially erased, which limits its use to applications where the data is infrequently updated.

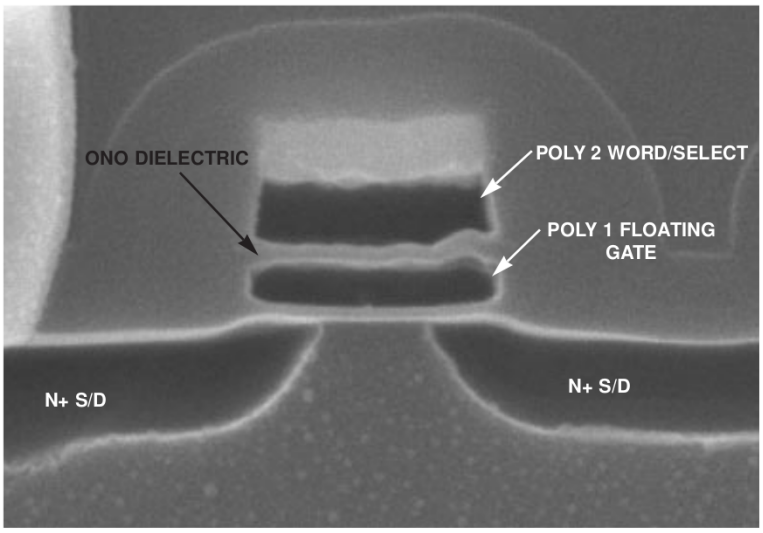

Figure 2.7: A cross section of an EPROM cell showing the floating gate [8].

EEPROM can be erased electrically, which allows the chip to be re-programmed in circuit. An EEPROM must be erased one byte at a time, which limits its write throughput. In order to increase write throughput, flash memory organizes EEPROM into blocks for erasing. There are two common flash architectures, NOR and NAND.

Figure 2.8 shows the equivalent circuit for NOR flash. In NOR flash, each transistor is directly connected to ground, and drives its value onto the bit line when selected by the word line. This direct connection enables NOR flash to provide and low-latency random read access and therefore directly replace EEPROM in many applications.

NAND flash was developed to be a denser alternative to NOR. NAND flash connects multiple transistors in series in order to reduce the area necessary for ground connections. This reduces the drive strength on the bit lines, however, which leads to longer read times. To mitigate this effect, NAND is organized into pages with the addition of access transistors. Reads are performed at the page granularity, which reduces the average read latency when locality is high. This makes NAND flash suitable for storage applications, in particular for media storage. Figure 2.9 shows the equivalent circuit for NAND flash. As Figure 2.10 illustrates, even with the addition of the page-access transistors, the amount of area required to implement NAND flash is greatly reduced by the removal of the ground contacts.

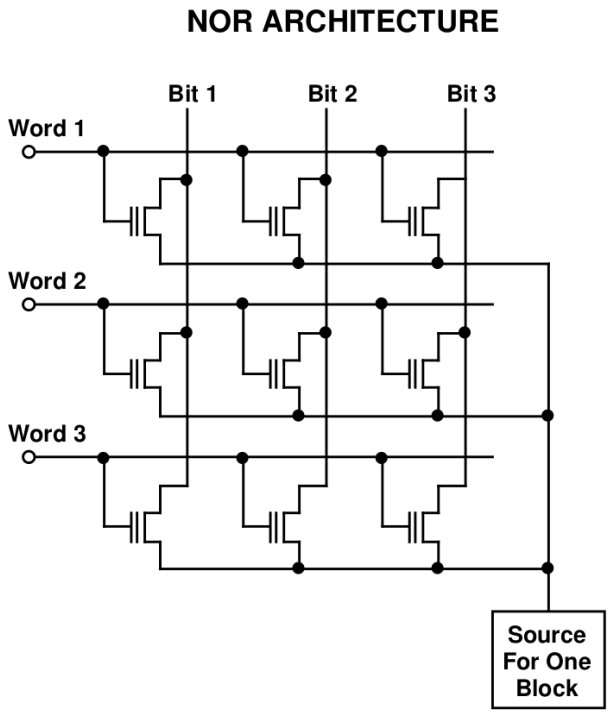

Figure 2.8: The architecture of NOR flash represented by the equivalent circuit [8].

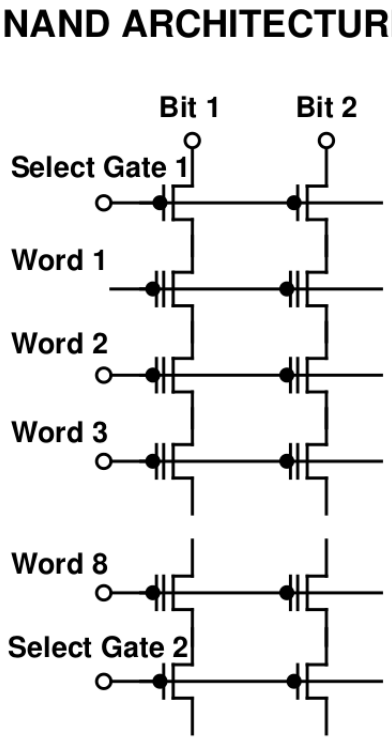

Figure 2.9: The architecture of NAND flash represented by the equivalent circuit [8].

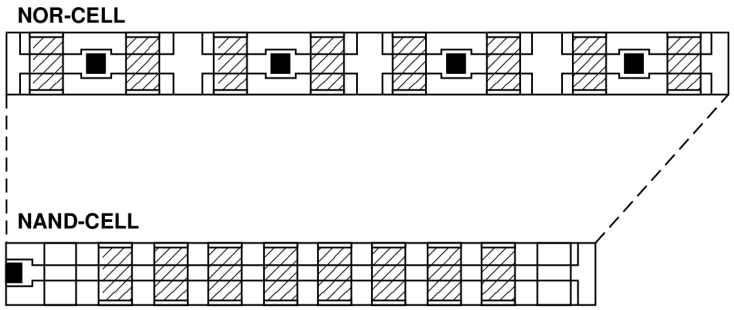

Figure 2.10: A comparison of the area required to implement NAND and NOR flash [8].

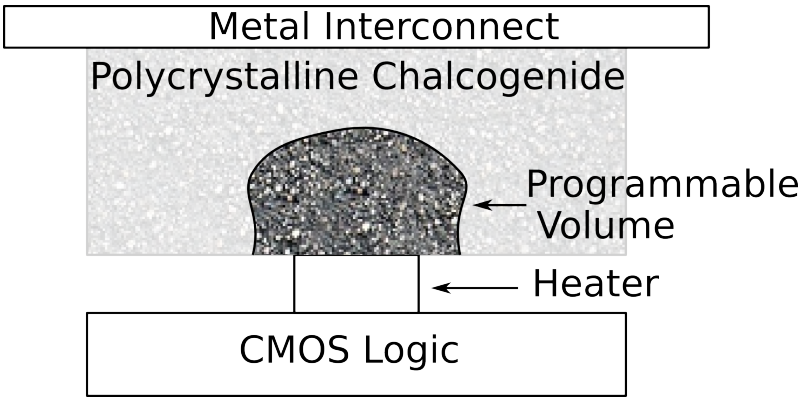

Figure 2.11: A PCM storage cell .

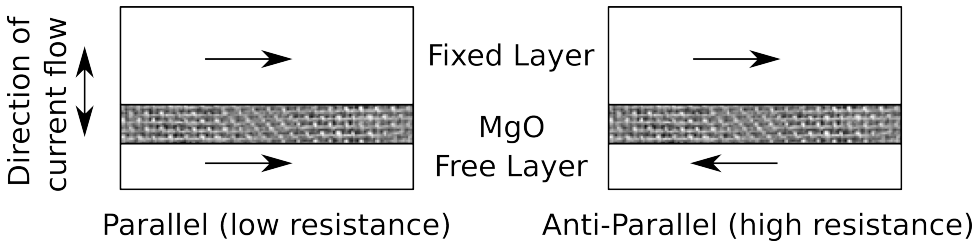

Figure 2.12: A Magnetic Tunneling Junction (MTJ).

2.3.2 Phase-Change Memory

PCM uses the dual states of certain chalcogenide compounds to store data. These materials have different crystalline states depending on the speed at which they are allowed to solidify. When they are cooled slowly, they form a crystalline structure. When they are cooled quickly, they form an amorphous solid. The crystalline form has a lower resistance, and this difference can be used for information storage.

PCM devices incorporate a mechanism for heating the chalcogenide, as shown in Figure 2.11. Heating it to its melting point resets the storage state of the cell. If no further heat is added, the device cools quickly into the amorphous state. If heat is added at the correct intervals, the device cools more slowly.

2.3.3 Magnetoresistive RAM

In a Magnetic Tunneling Junction (MTJ), illustrated in Figure 2.12, two magnetic layers are separated by a tunneling layer. One of the layers has a fixed magnetic orientation, while the other can be changed. The orientation of the free layer can be inferred by measuring the resistance to electric current passing through the MTJ. Electrons passing through the layers are scattered when their spins do not match the orientation of the magnetized layers. When the orientation of the free layer is parallel (P) to the orientation of the fixed layer, the resistance of the MTJ is lowest. When the orientation of the free layer is anti-parallel (AP) to the fixed layer, electrons of both polarities are scattered, which increases the resistance of the MTJ.

The orientation of the free layer in MRAM is changed by the magnetic field generated by a current passing through the wires. The current necessary to generate the switching field limits the scaling of this type of MRAM [9]. Figure 2.13 shows an MRAM cell.

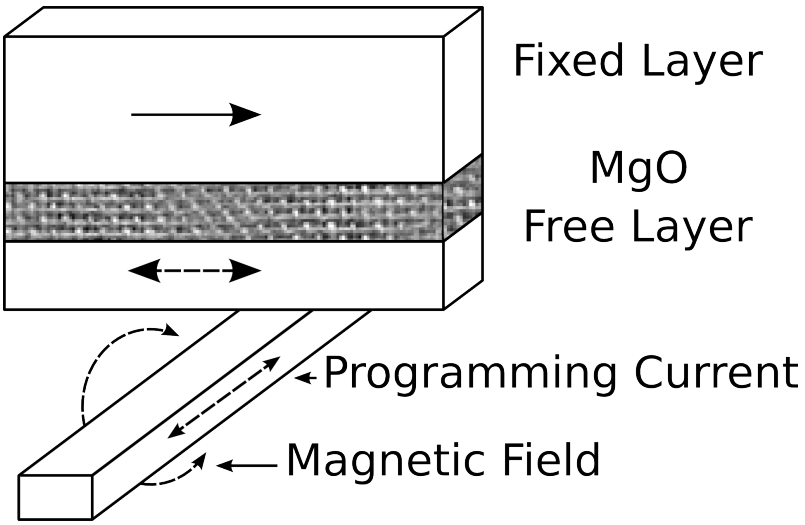

Figure 2.13: Part of an MRAM cell programmed with a magnetic field, showing the MTJ, the programming current, and the magnetic field.

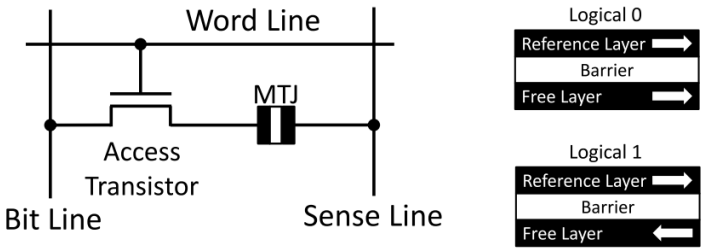

Figure 2.14: A STT-RAM cell [11].

2.3.4 Spin-Torque Transfer RAM

STT-RAM is a type of MRAM which utilizes the spin-torque transfer effect to lower the current necessary to change the orientation of the magnetic field in the free layer of the MTJ. If sufficient current is passed through the MTJ, the free layer changes orientation. When the current passes from the fixed layer to the free layer, the current is polarized, and the spin of the electrons forces the free layer to the same orientation as the fixed layer. When the current passes from the free layer to the fixed layer, the scattered electrons affect the orientation of the fixed layer and tend to force it into the AP state. Programming with polarized current is much more efficient than with scattered current, which creates asymmetry between the current requirements for AP to P and P to AP switching [10].

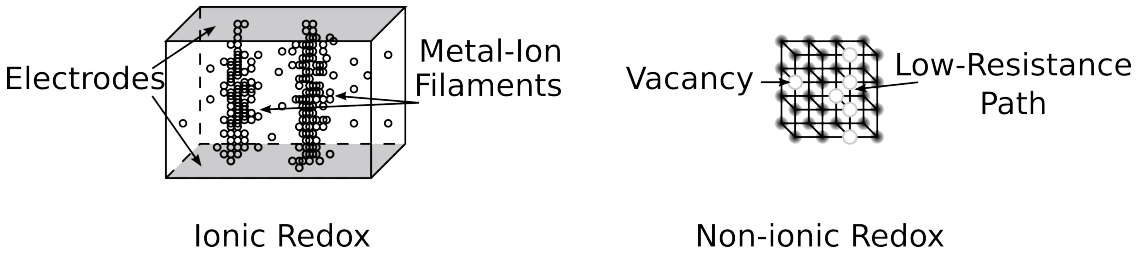

Figure 2.15: The two redox storage mechanisms used in RRAM.

2.3.5 Ferroelectric RAM

Ferroelectric RAM (FRAM), or FeRAM, is a non-volatile storage technology that is very similar to DRAM. FRAM is also built with one transistor and one capacitor per cell, but the dielectric in the capacitor is replaced by a ferroelectric material. Storing a charge on the capacitor causes a rearrangement in the crystal structure, which affects the current from the capacitor when the cell is read. Like DRAM, reading the state of an FRAM cell is destructive, so the cell must be rewritten each time it is read. Scaling for FRAM is also limited by the size of the capacitor.

2.3.6 Resistive RAM

RRAM, also known as ReRAM, is based on a circuit element with a low-resistance and a high-resistance state. The state of the device can be changed based on the voltage applied across the device. There are several types of RRAM, classified according to the mechanism by which their resistance changes. These include ionic redox RRAM and non-ionic redox RRAM, illustrated in Figure 2.15, and Correlated Electron RAM (CERAM).

In ionic redox RRAM, metal ions diffuse through a dielectric layer in response to an electric field. This ion migration forms or destroys conductive filaments, depending on the direction of the electric field. Whether or not these conductive filaments completely bridge the gap between the electrodes, they lower the resistance of the cell [12].

Non-ionic redox RRAM forms areas of high concentration of vacancies in the lattice structure of the insulator. These aligned vacancies create paths of lower resistance through the insulator, leading to a low-resistance state [12].

CERAM is based on a Mott-like metal-insulator transition in a Transition Metal Oxide (TMO), such as NiO. A Mott transition is an abrupt change in conductivity, and can be due to changes in pressure, doping, or temperature. In the case of

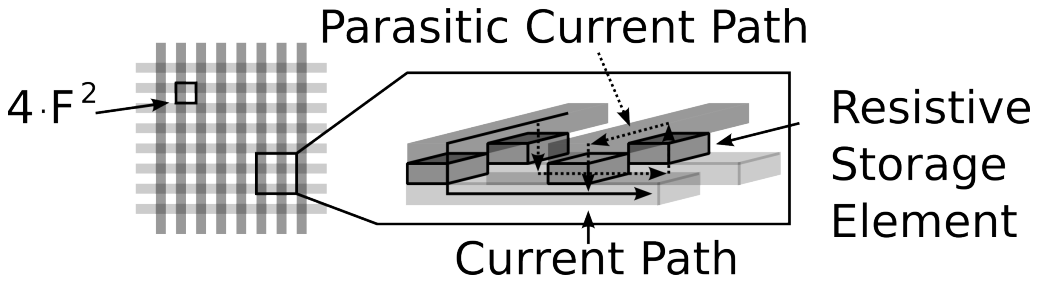

Figure 2.16: The top view of an RRAM array, showing how the density is determined by the minimum feature size, F, of the access lines. On the right is a small segment of the array, showing the desired current path and a parasitic current path.

CERAM, this transition is due to an electric field. Unlike redox RRAM, CERAM exhibits nonpolar switching, meaning that it can be set and reset with unidirectional or bidirectional electric fields [13], [14].

One of the benefits of RRAM is that it can be built in a crossbar-like architecture, shown in Figure 2.16, with the switching material connecting the cross points. In theory, the devices could be implemented without access transistors distributed in the memory array, allowing the RRAM arrays to scale with the minimum feature size of the technology. One challenge of this architecture is that leakage currents may flow through parasitic paths at other cross points, limiting the scaling of RRAM arrays. One proposed solution is the addition of diodes to limit parasitic paths.

2.3.7 Non-Volatile RAM Summary

There are many candidate technologies for non-volatile storage in future systems. The ideal candidate would approach DRAM in terms of read and write latency and approach (or surpass) NAND flash in terms of density and cost. Such a technology could then be used as a universal memory, and replace multiple storage technologies in the current memory hierarchy.

2.4 Scaling Faster Than Lithography

There are several technology-orthogonal techniques that can be used to increase the density of solid-state storage. These techniques include storing multiple bits in a single cell, increasing the number of process steps to add vertical layers of devices, and stacking multiple dice together.

2.4.1 Multi-Level Storage Cell

The storage technologies presented are based on being able to distinguish between two logical states when reading a value. Multi-Level storage Cell (MLC) is based on the idea of creating additional states, which enables the storage of multiple bits in a single storage cell. This requires tighter process control and more sensitive sense amplifiers. It may also require the addition of Error Correcting Code (ECC). This section discusses some aspects of MLC storage specific to DRAM, flash, and PCM.

MLC DRAM designs have been around for many years [15]–[17], including a 16-level MLC DRAM fabricated in 1987 [18]. Storing multiple states in DRAM has become progressively more difficult since the operating voltages have decreased, as noted in Section 2.2.2.

MLC flash is commercially successful in products that favor density or cost over performance. In MLC flash, the number of electrons trapped on the floating gate is subdivided to form more logic levels, often four [19]. With four logic levels, each transistor can store two bits of information. This nearly doubles the storage density, at the cost of longer read and write times and lower reliability. The reliability of MLC flash can be improved by the addition of ECC circuitry.

Four-level PCM requires ECC for reliable operation. In order to create a MLC PCM without ECC circuitry, one state can be removed. This yields a cell that stores ternary instead of binary information and increases the guard band between the remaining states. Two ternary cells can be combined to encode three binary bits of information, which is sometimes called 3-on-2 encoding [20], [21]. Since two ternary cells can encode 9 values, and three binary bits can only encode 8 values, the extra value can be used to further improve reliability or speed encoding and decoding.

2.4.2 Vertical Process Extensions

Another method for increasing density is the vertical extension of fabrication technologies. This can be accomplished using epitaxial growth of another layer of silicon, chemical polishing, and repeating fabrication steps for the following layers [22]. It can also be accomplished by etching deep trenches, which are then filled to form vertically interconnected transistors [23]. These methods differ in their implementation costs and theoretical vertical scaling limits [24]. Both methods require additional mask steps, which increase the start-up and per-wafer fabrication costs. Although extending the process vertically can also be considered 3D, it should not be confused with 3D integration, which involves vertically interconnecting multiple dice or wafers.

2.5 Memory Conclusion

The search for a universal memory technology has been ongoing for decades. There are several promising technologies in terms of density, latency, bandwidth, and volatility. Many of these technologies have been scheduled to be available in the next few years for many years. DRAM and flash memory technologies have continued to improve more quickly than new memory technologies. One way to reduce the risk of missing the transition to a new memory technology is to adopt a memory interface which is technology independent.

3 Simplifying the Memory Hierarchy with DiskRAM

This chapter presents DiskRAM, a prototype implementation of a memory controller with hardware swapping. It starts with background information about storage hierarchies, discussing segmentation, paging, and swapping. It then presents related work and motivates the approach used in DiskRAM. After presenting the DiskRAM design and implementation, it describes the difficulties encountered while debugging the system and the current state of the design.

3.1 Background

DiskRAM is a hardware prototype memory controller. In order to understand the advantages of the features that DiskRAM provides, it is important to understand the underlying concepts. This section briefly reviews storage hierarchies and their history.

3.1.1 Storage Hierarchies

Storage hierarchies were invented early in the history of computer system design, to allow designers to incorporate multiple memory types into a single system. Storage hierarchies are needed when there are multiple memory types that are differentiated by price and speed. For early machines, vacuum tube storage was fast and expensive, while magnetic drums were slower and relatively cheap. A recent machine with a deep storage hierarchy could include SRAM caches for the CPU, DRAM main memory, solid state disk drives (flash), hard disk drives, and magnetic tape. At each level of the hierarchy, the cost per byte decreases and the access latency increases. Figure 3.1 presents a sampling of cost and performance data for SRAM, DRAM, and disk storage.

Memory hierarchies are designed to approximate the performance of a more expensive storage technology for the price of a less expensive technology. This is

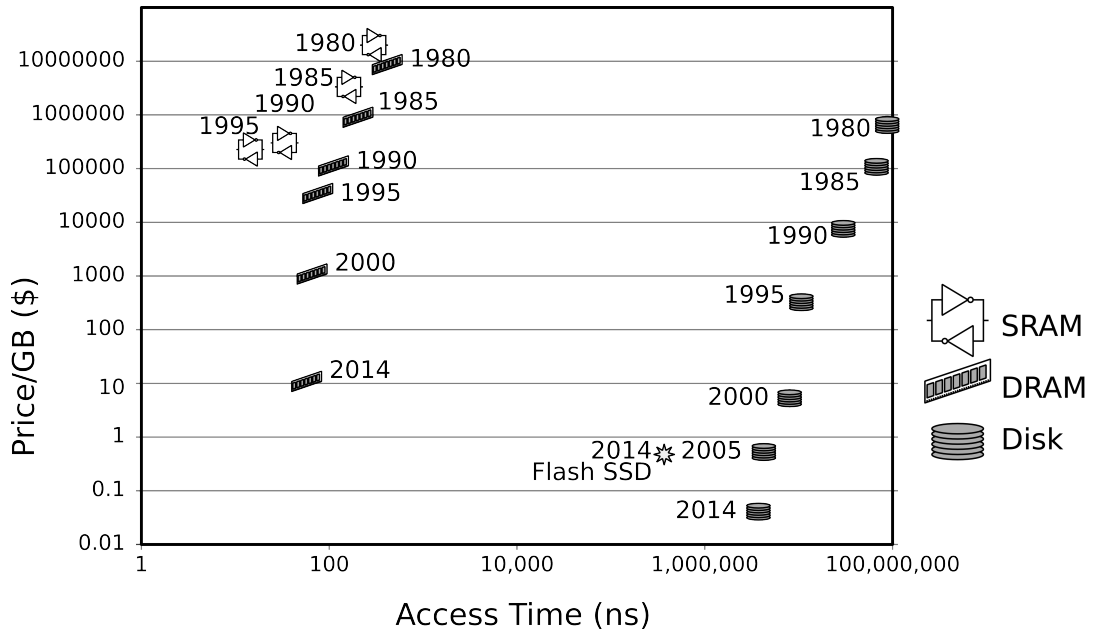

Figure 3.1: Sampled price and performance points for SRAM, DRAM, and disk. Data points before 1996 from Hennessy and Patterson [25]. Note that the SRAM in the memory hierarchy is integrated in current microprocessors.

possible since the amount of data which is needed at any one time is much less than the total amount of storage in the hierarchy. Which data is stored in each level of the memory hierarchy determines its performance. This decision can be made in hardware, for example in processor caches, or in software.

3.1.2 Segmentation, Paging, and Swapping

In order to discuss software management of the storage hierarchy, some terms must be defined. This section describes segmentation, paging, and swapping.

Aside from performance, other reasons may exist for creating mechanisms to manage memory hierarchies. The limited register size in early microprocessors, for example, was partly compensated for using segmentation. Segmentation increases the size of the physical address using a pair of registers that hold the upper address bits for the current code and data segments. Before the address is driven onto the bus, the upper bits are added to the address, according to the type of request. Current 64-bit processors have large enough registers that segmentation support is only included for backward compatibility; most modern operating systems and applications configure the segmentation registers to provide a flat(unsegmented) address space.

Paging is another memory management mechanism. It can be used to provide

isolation between processes and support swapping and caching. Paging is based on the idea of virtual memory [26], or separating the concept of the address where data is stored, the page frame, from the data itself. In other words, an extra level of indirection is added which maps virtual addresses to the physical address where the data is stored. One of the benefits of virtual memory is that processes can be compiled to use the same virtual memory address, and then be loaded into separate physical addresses.

When memory is accessed, the virtual address is mapped to a physical address. Virtual to physical mappings are stored with access permissions and usage bits in page tables in memory. These page tables are stored hierarchically in pages. If there is a matching entry, the upper bits of the virtual address is replaced by the virtual address, and the access continues. In order to avoid searching through the page tables for each request, recently used mappings are stored in a Translation Look-aside Buffer (TLB). Page tables must then be searched once for each mapping miss in the TLB.

Pages can theoretically be any size. Some common sizes are 4KB, 8KB, 2MB, and 1GB. Larger mappings decrease the number of mappings, page misses, and TLB entries, but increase the amount of unused memory, or fragmentation. Since the TLB must be checked for every memory reference, it is desirable to keep it as small and as fast as possible.

If no mapping for a virtual address exists in the page tables, a page fault occurs. This can be due to a programming error or swapping. When the total working set size is larger than the amount of memory available, page faults can occur when the requested data is stored in the next level of the storage hierarchy. The operating system can then bring the page into memory and update the page tables to reflect the new mapping. This operation is referred to as swapping.

3.1.3 Performance Implications

In order to achieve good performance, the amount of swapping must be kept to a minimum. When the operating system removes pages that will soon be needed, thus causing more swapping, it is said to thrash. Thrashing can reinforce itself and quickly render a system unusably slow. In the extreme case, every memory reference causes a page table walk and then initiates data transfers to and from a disk. As pointed out in Figure 3.1, disk accesses are thousands of times slower than accesses to memory.

One of the reasons that thrashing is so problematic is that the operating system decides which pages to evict based on usage information in the page tables. When a mapping is brought into the TLB, a bit in that page-table entry is set. In order to keep the page-usage information current, the operating system periodically clears the usage bits and flushes the TLB. Thrashing reduces the number of pages which are accessed per unit time, which reduces the number of page tables with their usage bit set. When most of the pages are unused, the choice of which page to evict essentially becomes random.

3.1.4 Programmer Perspective

Thrashing presents a dilemma for application programmers since it is largely out of their control. Applications may be run on a variety of platforms, with varying amounts of RAM, and concurrently with other tasks. In order to help the operating system make more informed decisions, applications with regular memory-access patterns sometimes resort to managing their own memory. These applications include those for image manipulation, database management, and video editing. In order to perform their own memory management, they restrict the amount of memory that they use and explicitly transfer portions of their working set into a memory buffer.

To see why this helps the operating system make more informed decisions, imagine a program which processes large streams of data from files. The simplistic approach would be to load the entire file into memory before processing. When sufficient memory is available, this is an effective method. When memory is scarce, portions of the file which have not yet been accessed may be removed from memory since the operating system favors recently accessed pages. Aside from the current page, pages which have been accessed are less likely to be reused in streaming data. Allocating a memory buffer into which the file is copied reduces the amount of memory which is required and changes the access pattern, from the operating system's point of view, so that the pages are frequently reused.

One of the problems with this approach is that it is difficult to correctly size the buffer. The optimal size of the buffer is the minimal size that is large enough so that the processor does not have to wait for the disk. This is dependent on the relative speeds of the processor, the memory, and the disk. If any of these parameters change, or if other programs influence their performance, the optimal value changes. In other words, although virtual memory simplifies the memory hierarchy, its performance characteristics are more complex and harder for the programmer to control.

3.2 Simplifying the Memory Hierarchy

The goal of this section is to motivate the creation of a memory hierarchy which abstracts the details of the underlying storage. It describes the desired abstraction, the need for accurate page usage information, related work, and some of its possible benefits.

3.2.1 Storage Abstraction

Storage abstraction does not mean that the programmer is freed from optimizing the working set size and access patterns of software. The purpose of abstraction is to free the programmer from the details of storage management. This can be split into two related subgoals: the programmer doesn't need to know about the underlying storage technology and the software scales with storage.

When thrashing was discussed in Section 3.1.4, some of the difficulties of memory management were presented. Most of these are only known at run time, including the amount of resources available and the number of other programs which are running. The goal is to abstract these details away, or hide them from the programmer, to simplify code development.

A good implementation of this abstraction allows the software to scale with the storage available. When software is run on larger machines or with more available memory its performance should increase. Another way to think of it is that its performance should degrade gracefully. This requires an efficient allocation of each level of storage among the currently running tasks.

3.2.2 Accurate Page Usage Information

Accurate page usage information is necessary in order to make good run-time decisions on page allocation. Collecting memory usage information is expensive because of the frequency of memory accesses, and the fact that memory accesses are often on the critical path for execution. Current processors eliminate much of this overhead by only keeping an access bit in the page tables, as discussed in Section 3.1.3. Using this method, it is relatively straightforward to increase the accuracy of the page usage information which is collected: the solution is to flush the TLB and reset the access bits more often. The drawback is the decreased performance due to the time taken for TLB misses and page table walks.

3.2.3 Hardware-Controlled Page Management

Prior work used fine-grained page-usage information to calculate a miss-ratio curve [27]. Overheads of 7%-10% were reported, depending on the application, for collecting this information in the operating system. Using this information to alter the memory allocation significantly reduced the response time of memory-bound interactive applications with only minor effects on the performance of the non-interactive applications. Collecting the information in hardware reduces the overhead involved.

3.2.4 Single-Address-Space Operating Systems

Some operating systems avoid the problems of virtual memory altogether. One of the most successful of these is the operating system of the IBM System 38, which later became the AS/400 [28]. This system viewed all storage as flat and relied on a virtual machine to manage the transfers between system memory and disk. This extra level of indirection simplified the migration of application between physical machines with different amounts of physical memory.

3.3 Prototyping

In order to investigate the advantages and disadvantages of hardware-controlled paging, a prototype called DiskRAM was built. The following sections explain the purpose of building prototypes instead of simulating, and how the performance of the prototype should be measured. It then details the process of building the prototype, along with some of the successes and failures encountered.

3.3.1 Why Not Simulate?

An alternative to prototyping the proposed memory hierarchy is to characterize it through simulation. Simulation is attractive because systems can be simulated at various levels of abstraction, all aspects of the system are configurable, and they can be run on commodity systems. As the size of the simulated system scale, the difficulties of full-system simulation make prototyping a more attractive option [29]. Some of the difficulties of simulating a complete memory hierarchy are simulation slowdown, the size of the memory hierarchy, the size of reasonable workloads, and the number of parameters.

Full-system simulators typically exhibit a slowdown of several orders of magnitude,

depending on the level of detail at which they are run. This is not surprising, considering the complexity of the system and the number of interacting components. Accurately simulating a single memory reference involves simulating multiple levels of caches, a TLB, and a DRAM controller. This becomes even more complex when paging to disk. In order to achieve reasonable simulation rates, simulators raise the level of abstraction for some components, while focusing on the performance of others.

In addition to the slowdown, another issue is that the amount of storage required for simulation is larger than the amount of storage available on the simulated architecture. This is problematic since future systems generally have more storage than current systems. This significantly increases the cost, complexity, or slowdown associated with full-system simulations.

One technique used to cope with the slowdown and increased storage requirements of simulation is to reduce the size of the workload. For full-system simulation, it is very difficult to reduce the size of the workload while preserving the behavior of the modeled memory hierarchy. This is compounded by the fact that the size of interesting workloads increases over time to take advantage of successive hardware generations.

Another difficulty of using full-system simulation for memory hierarchies is the number of free parameters in a simulation. Incorrectly specifying any timing parameter for any one of the components can affect the accuracy of the simulation. At best, this leads to skewed results that misrepresent the performance difference due to the proposed architectural change. Far worse, incorrect parameter choices can potentially change the outcome. Sensitivity studies can be useful to understand the effect of parameter choices at the cost of increasing the number of simulations which must be run. Creating a functional prototype allows the designer to increase the level of abstraction for the components that are implemented in hardware, and constrains many of the free parameters.

3.3.2 Performance Comparisons

Although the performance of a hardware prototype is significantly better than that of simulation, it may still be slower than commercial systems. This can be due to the lack of optimization due to the nature of prototyping or the use of hardware, such as Field Programmable Gate Array (FPGA)s, to increase the flexibility of the prototype. In order to factor out the slowdown due to the prototype and focus on the proposed architectural improvements, the prototype should be compared against

itself in different configurations. This minimizes the number of variables and makes the comparison as fair as possible.

3.3.3 Prototype Design

Once the decision is made to create a prototype, the design space must be explored. Modeling DiskRAM requires a programmable memory controller, which has access to all of the memory references in a system. Figure 3.2 shows a simplified block diagram of a multiprocessor system and highlights three of the interfaces which could be used for prototyping a memory controller: the interface between the processor and RAM, the interface between the processor and the chipset, and the interface between processors.

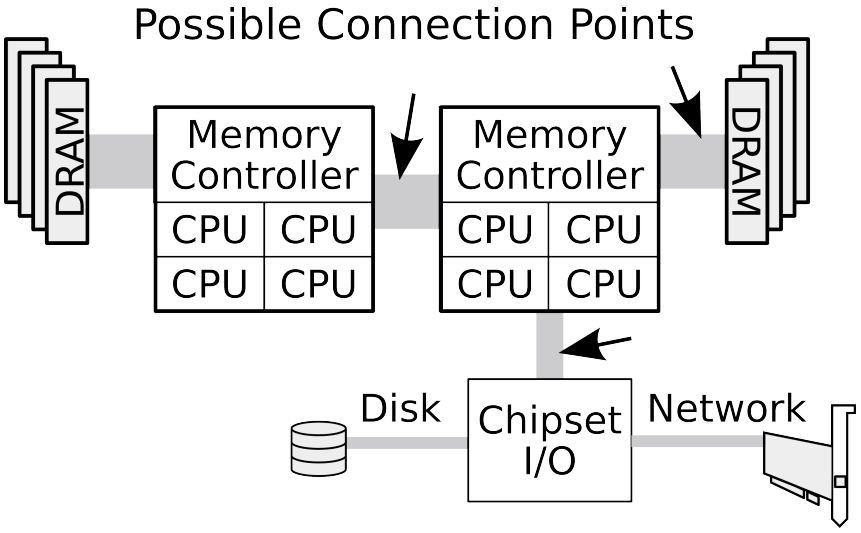

Figure 3.2: A simplified block diagram of a multiprocessor system highlighting the possible connection points for prototyping.

Prototyping using the interface between a processor and the RAM is tempting because the interface is well documented, provides high bandwidth, and is supported by FPGA vendors. Unfortunately, the RAM interface is inflexible. It is limited in the number of address bits which are available and has fixed timing. This means that although the processor could write to an FPGA, special support must be developed to allow arbitrary reads.

The interface between the processor and the I/O device is also interesting since access to that interface provides access to disk and other I/O devices. A possible problem with this interface is whether the processor allows memory accesses to be routed to the I/O devices and how they are modified.

The third possible interface is the interface between the processors. This interface is promising since memory references from one socket to the other and cache coherency traffic between the CPU sockets traverse it. One downside is that coherent interfaces are proprietary, which increases the difficulty of prototyping with them and limits the amount of information that can be published.

3.3.4 HyperTransport

Connecting to the processor as an I/O device is the easier of the two viable options. Since HyperTransport connects AMD Opteron processors to I/O, it was chosen for the basis of the prototype. HyperTransport is an open standard from the HyperTransport consortium that is software compatible with Peripheral Components Interconnect (PCI), making it easier to find software and firmware that support it. Another benefit of HyperTransport is the availability of an open-source controller from the University of Heidelberg [30].

Once HyperTransport is selected as the interconnect, there are two commercially available methods to attach to the processor. One is to use an HTX board from the Unversity of Heidelberg [31]. The HTX slot was designed to enable low-latency I/O with Hypertransport, and several commercially available motherboards included a HTX slot. Unfortunately for this project, the card has no integrated DRAM or storage-access possibilities. The other option is an in-socket accelerator. DRC Computer and XtremeData have built FPGA boards that are pin compatible with an Opteron Processor [32]. Both provide access to the HyperTransport links and the DRAM interfaces of the processor. The main difference is that the DRC Computer offering is based on a Xilinx FPGA, while XtremeData uses an Altera FPGA. For the purposes of this work, either would be fine, and the XtremeData XD100 was chosen. A block diagram of the dual-processor motherboard that is shipped with the XtremeData XD1000, the Tyan Thunder K8SE (s2892), is shown in Figure 3.3.

As previously mentioned, using an FPGA slows down the performance of the system since it limits the maximum attainable frequency of the HyperTransport interface. The HyperTransport link between the Opteron processors on the s2892 runs at 800 MHz, or 1600 MT/s since it is Double Data Rate (DDR). With the XD1000 FPGA installed, the HyperTransport link is limited to 400 MHz.

The limitation in frequency is not the only performance limitation. Since the XD1000 appears to be an I/O device, its memory mapping affects its peak performance. Simple latency and bandwidth measurements reveal that the memory mapping has a greater effect than the frequency limitation [33]. Since relative perfor-

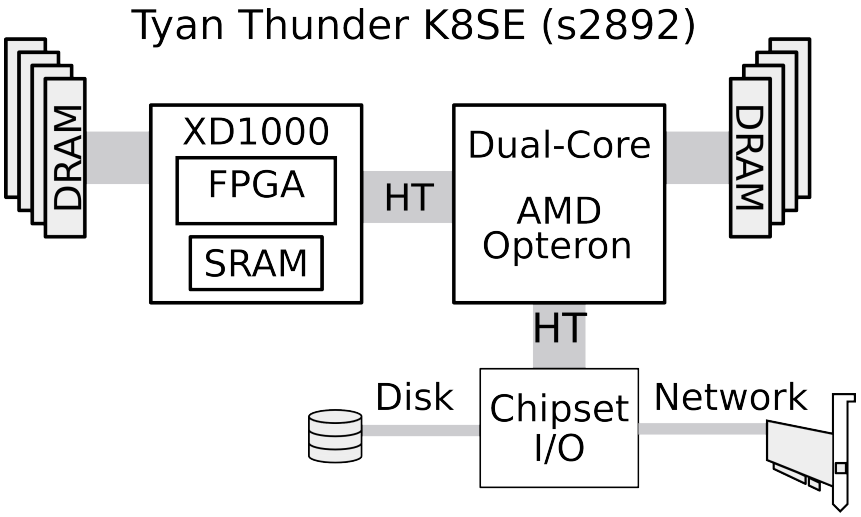

Figure 3.3: The block diagram of an XtremeData FPGA accelerator in a dual-processor AMD mainboard.

mance measurements are used, the actual performance numbers are less important.

3.3.5 Deadlock

Using the s2892 with the XD1000 to prototype a memory hierarchy introduces the possibility of creating a deadlock. Figure 3.4 shows the flow of packets that lead to a deadlock scenario, explained in the following steps:

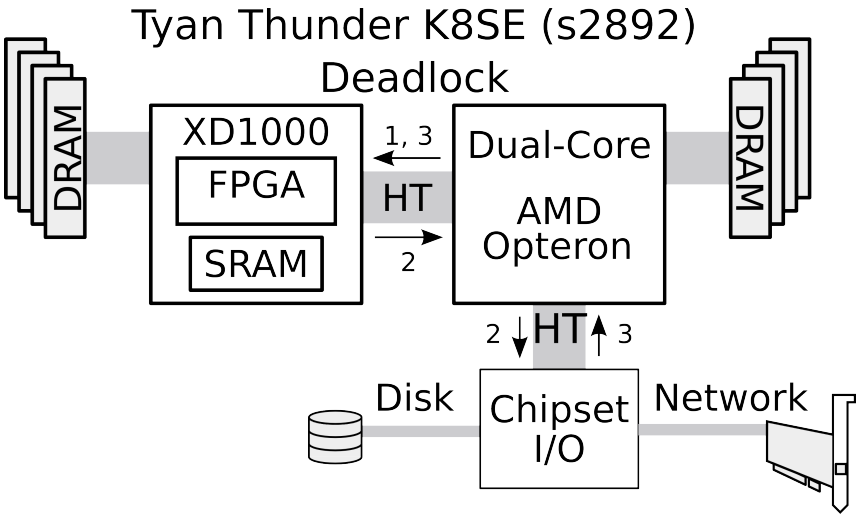

Figure 3.4: The deadlock scenario using an XtremeData XD1000 to prototype memory hierarchies in the s2892. The packet types are numbered as follows: 1 - CPU requests, 2 - XD1000 requests to disk, 3 - data from the disk.

1. The CPU sends a write request (1) to the XD1000 which can not be serviced from RAM.
2. The XD1000 initiates a transfer from the disk (2).
3. The disk sends the data back to the XD1000 by way of the CPU (3).

Since the XD1000 can not complete the write requests without receiving the writes from the disk drive, a deadlock is created. Several possibilities were explored in order to remove the deadlock, including using reads from the XD1000 instead of writes from the disk. This was unsuccessful since the pass-posted-writes bit was not propagated at some point in the chain. In the end, the decision was made to use a different mainboard to break the deadlock cycle.

3.3.6 Dual Chipset Motherboard

The Tyan Thunder K8WE (s2895) mainboard can be used to remove the deadlock since each CPU socket has its own connection to the chipset. Figure 3.5 shows the new flow of packets for write requests which cause disk activity. Note that each packet type travels in a different physical channel, removing the cycle and preventing deadlock.

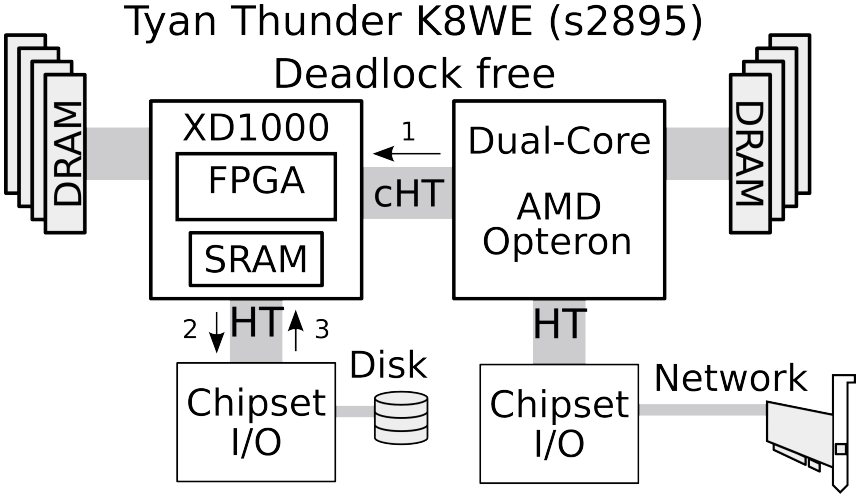

Figure 3.5: The block diagram of an XtremeData FPGA accelerator in a dual-processor AMD mainboard with two HyperTransport connections to I/O. The packet types are numbered as in Figure 3.4: 1 - CPU requests, 2 - XD1000 requests to disk, 3 - data from the disk.

Using the s2895 instead of the s2892 requires major modifications to the design. Two HyperTransport cores need to be included in the XD1000, and the changes to the BIOS needed to be ported. This opened up the possibility of additional changes,

including using the coherent HyperTransport core which is also developed by the same group [34], now at the University of Heidelberg, to connect to the CPU. The coherent HyperTransport core is only available from AMD under a non-disclosure agreement, but the authors of the coherent HyperTransport core provided contacts to AMD. They were excited to hear about the project and provided the NDA and access to the core and its documentation. In Figure 3.5, this change in connectivity is represented by cHT, which replaced the non-coherent connection from figures 3.3 and 3.4. This significantly increases the read bandwidth to the XD1000 [33].

3.3.7 BIOS Modifications

The XD1000 development system comes with a version of LinuxBIOS (now Coreboot), which has been modified to initialize the extra HyperTransport link. The XD1000 appears to be a PCI bus, which must be enumerated to allow the driver to be associated with the device. In addition, the FPGA clocking circuitry requires longer than an Opteron to stabilize, so some extra reset logic must be implemented to make sure that the XD1000 responds to configuration requests from the BIOS. These changes are the minimum that must be ported to the s2895. Using the coherent HyperTransport interface between the XD1000 and the CPU requires further changes.

The BIOS expects all coherent HyperTransport devices to be AMD Opteron CPUs. In order to initialize the system correctly, the XD1000 is added to the list of supported coherent devices. Device-specific initialization functions are implemented for memory allocation, managing the routing tables for memory requests, and enumerating the PCI devices attached to the XD1000.

3.3.8 Serial ATA

For the DiskRAM implementation, the most important I/O device attached to the XD1000 is the Serial ATA (SATA) controller. SATA was chosen since it allows a wide range of disk drives and solid-state drives to be attached. A well-supported PCI Express (PCIe) SATA card with an open-source Linux device driver was chosen. The card implements the AHCI specification, which was developed by Intel, and combines Parallel ATA commands into blocks to increase the efficiency of command transfers from the host. Using the device driver to initialize the SATA card simplifies the controller logic implemented in the XD1000. Once the card is initialized, block read and write commands can be implemented with a small number of posted writes. This is implemented as a simple state machine in the storage controller.

Supporting reads and writes from the XD1000 as well as BIOS and driver initialization requests from the CPU requires some extra logic in the HyperTransport interface controllers. Requests are separated based on their destinations and addresses, as well as the packet type.

3.3.9 Cache Controller

Once the HyperTransport controllers, SATA controller, and BIOS modifications are complete, all of the components necessary to implement DiskRAM are in place. Figure 3.6 presents the block diagram for the FPGA design. As an initial implementation, a simple set-associative cache with up to four ways was constructed. The cache is built using the SRAM included in the XD1000 as tag RAM.

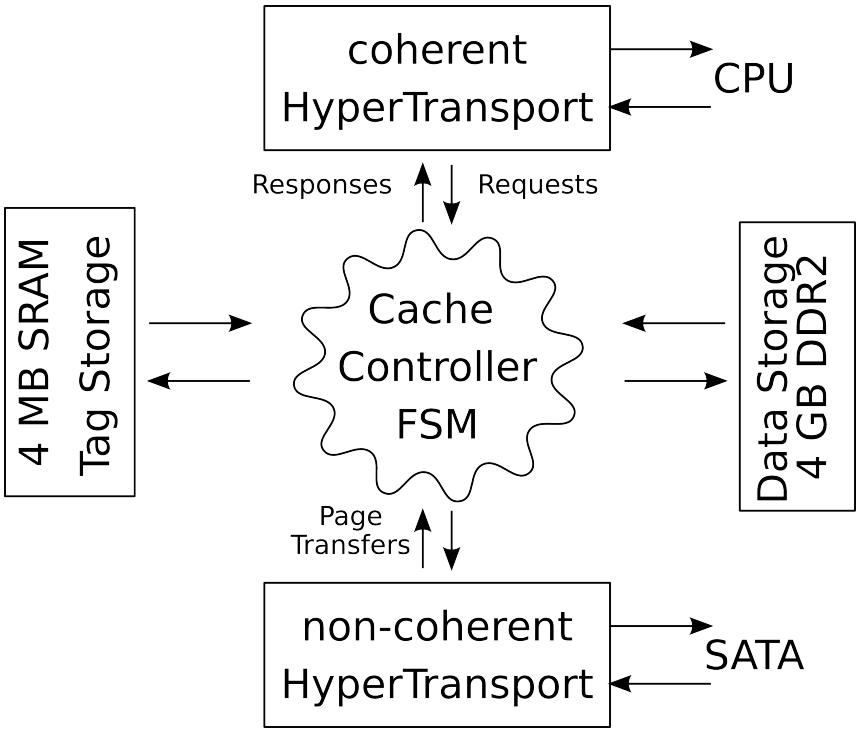

Figure 3.6: The block diagram of DiskRAM, showing the two HyperTransport interfaces, the DDR2 interface, and the SRAM interface, which is used for tag storage.

When a request is received from the CPU, the address is partitioned as shown in Figure 3.7. The index is used to retrieve the possible tags from the SRAM. If one of the tags matches, the request hits in the cache, and the data request is completed by the DRAM controller. If not, the block of memory is retrieved from the disk, the tags are updated, and the request is served as if it were a hit.

The coherent HyperTransport core supports a 40-bit physical address range, or up

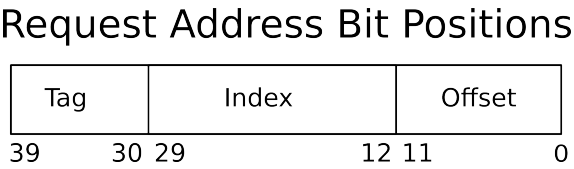

Figure 3.7: An illustration of how the address of a CPU request is used for cache lookups in DiskRAM. This example is for a 4-way set-associative cache with 4KB blocks.

to 1 TB. This is the maximum size of the storage addressable by DiskRAM. Since the XD1000 supports 4 GB of DDR2, and provides 4 MB of SRAM, a block size of 4 KB was chosen. This fits well with the page size of the operating system and thus minimizes fragmentation. It also provides a 32-bit tag in SRAM for each cache block. The tags are used to store the state of the block, the tag, and the usage information, as shown in Figure 3.8. Unlike processor caches, which have state bits for sharing, the cache only needs to store valid and dirty bits since there is only one cache in DiskRAM.

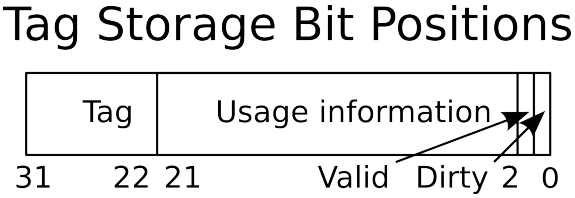

Figure 3.8: A 32-bit tag entry divided into tag address bits, valid and dirty bits, and usage bits. This example is for a 4-way set-associative cache with 4KB blocks.

3.3.10 Testing

DiskRAM is a complex system with many components. As with any complex system, testing must be performed to find and remove bugs that were introduced in the implementation. DiskRAM was tested in three ways: using simulation, using memtest programs from the BIOS, and booting unmodified operating systems.

For simulation, the BFM from XtremeData was extended to include support for coherent HyperTransport packet types, and a model for the SATA card was developed. Random request streams were generated, and the results were checked by parsing the simulation log and running it through a cache simulator written in Ruby.

When no more errors were found through simulation, DiskRAM was implemented for the XD1000 and installed. The first step was to modify the BIOS to initialize DiskRAM, including the memory and attached disk, so that it appeared to the processor as RAM. Once the hardware was initialized, a memtest function written in

C was used to test it. Before the test was started, the disk was patterned to verify that the correct blocks were being read. These tests were run with and without caching enabled, in order to test different block sizes and commands.

At this point, two unmodified operating systems, Ubuntu [35] and DSL [36], were booted. In order to force all memory requests to be serviced by DiskRAM, the RAM connected to the CPU was removed. The resulting system is shown in Figure 3.9.

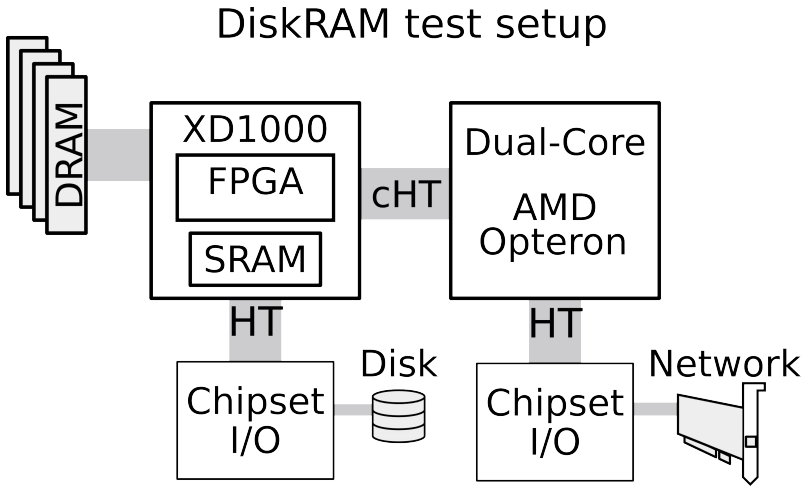

Figure 3.9: The system from Figure 3.5 modified for the OS boot tests. The RAM connected to the CPU has been removed so that all memory requests are served by DiskRAM.

The Ubuntu system boots until the hard drive is initialized, and then the system hangs. At this point, there is no obvious error in the state of DiskRAM. There appears to be some non-determinism in the failure, and it is hard to tell if this is due to non-determinism in the timing of disk responses, or due to a race condition in DiskRAM. Significant effort has been invested into isolating the failure, but it has not been found. At this point, simplifying the system to test a middle ground between the memory tests and Ubuntu seems promising.

DSL is a smaller distribution of Linux, and hopefully simpler. The DSL system boots into text mode, but in graphical mode, there is some corruption in the display. Again, there are no obvious errors in the internal state of DiskRAM. The corruption is also not completely deterministic.

One of the keys to successful debugging is finding the simplest system configuration that exhibits a failure. Using a simple configuration as a starting point reduces the debugging effort and increases the chance of finding and fixing the problem. When simplifying the system is infeasible, there are other techniques that can lead to progress.

3.3.11 The Path Forward

Some of the things that could make a difference in the ability to debug DiskRAM are documentation, verification, and more insight into the state of the other components. In general, having access to more information should make a significant difference in the effort required to test and debug DiskRAM.

In the systems used to test DiskRAM, there is a significant number of components besides the XD1000 FPGA. The Opteron processor and the two NVIDIA nForce southbridges are the most important of these. Unfortunately, documentation for these components is available only with a non-disclosure agreement, which is difficult to obtain. Section 3.3.5 describes the deadlock problem with the Tyan s2892 board, a problem that was much more difficult to solve without the proper documentation. If the documentation for both components had been available, it would have been clear which component ignored or removed the HyperTransport bit that allows responses to pass posted writes. Knowing that the deadlock could not be solved in this way would have allowed the decision to move to the Tyan s2895 board to be made earlier, saving time.

The lack of documentation also makes it more difficult to perform system verification. Although the components of DiskRAM could be individually modeled in a verification environment, more documentation would be needed to ensure sufficient coverage.

Along with documentation, in-system debugging tools would be helpful. The ability to halt the system at various stages of booting to check the state of the processor and southbridge would make it possible to see if unusual bus activity is occurring, if the devices detected error conditions, or if flow control problems had occurred, e.g. missing or extra credits.

More information, especially more documentation for the processor and the southbridge chips, would have been helpful in speeding up the testing and debugging of DiskRAM.

3.4 Conclusion

Coherent HyperTransport can be used as a packetized memory interface for experimentation with the memory hierarchy. This chapter described DiskRAM, a project designed to emulate a simplified storage hierarchy using commodity hardware and an FPGA-based accelerator. DiskRAM provides hardware support for swapping

and allows this functionality to be removed from the operating system. In order to implement DiskRAM, a cache controller and a SATA device driver were implemented and integrated with the DDR2 and SRAM controllers, which come as part of the XD1000 development kit.

Although DiskRAM passed the functional tests, it was unable to boot unmodified operating systems. This can be attributed to bugs in the implementation of DiskRAM. Although bugs were found and fixed in the implementation, the process of debugging was slow. In order to debug DiskRAM further, more information and equipment is necessary.

The next chapter presents another pathway to experimenting with memory hierarchies using packetized interfaces, the Network-Attached Memory. One of the benefits of using a network interface is that the number of components which are interacting with the development system is reduced, which should reduce the effort needed for implementation and debugging.

4 Network-Attached Memory (NAM)

This chapter presents Network Attached Memory (NAM), a project which uses a packetized network interface for exploring the addition of global storage to the memory hierarchy in a distributed system. The network interface provides the ability to tunnel arbitrary commands to a prototype device, in addition to any native support for read and write commands, which makes it a flexible interface for prototyping.

The idea for NAM came from the desire to provide high-performance checkpoints in the DEEP-Extended Reach (DEEP-ER) project. This chapter provides background information about the Dynamical Exascale Entry Platform (DEEP) and DEEP-ER projects and explains the checkpoint/restart mechanism, including some of the difficulties which arise when trying to scale it to exascale. The NAM architecture is presented as a way to ease the implementation of a high-performance checkpoint system, and as a way to increase the performance of generic I/O operations, such as writes to a parallel file system.

4.1 DEEP and DEEP-ER

DEEP is a European Union project funded under the Seventh Framework Programme [37]. Its mission is to develop a computing architecture which will allow systems to scale to exascale. The DEEP architecture is based on the separation of a super computer into two main parts: the cluster and the booster [38]. The cluster consists of commodity compute nodes with a commodity interconnection network. The booster nodes are designed especially for DEEP and are based on the Intel Many Integrated Cores (MIC) architecture. Since the booster uses a 3D torus direct network topology, the cost of the interconnect scales linearly with its size. Separating the cluster from the booster allows the parallel and sequential portions of the machine to scale independently.

DEEP-ER is the follow-on project to DEEP [39]. It aims to improve the usability and the reliability of DEEP hardware using updated components. Much of the focus for the project is on I/O and checkpointing system design and performance. These

problems are very related since the checkpointing system relies on I/O performance. Evaluation of the bandwidth available in the DEEP architecture showed that a level of hierarchy could be used to bridge the gap in available bandwidth from the nodes to the parallel file system (PFS).

4.2 Reliability of Exascale Systems

It is generally accepted that the mean time to failure for Exascale systems will be shorter than for Petascale systems since the component count is projected to increase and the failure rate of each component will not improve sufficiently to counteract this [40]. Increasing failure rates reduce the efficiency of the system and, if they are too high, render it unusable. This section discusses checkpointing and its role in improving the reliability of systems, including multi-level checkpointing, and suggests adding a Network-Attached Memory (NAM) to the system.

4.2.1 Checkpointing

One way to mitigate the failure rate is by regularly saving the state of the computation. When a component failure causes a node failure, the computation can be restarted if the prior state is known. These saved states are called checkpoints. Checkpoints can be system level or application level. System-level checkpoints save the entire state of the machine, so that computation can begin exactly where it left off. Application-level checkpoints save the minimal set of data needed to resume, which requires less storage but more implementation effort.

4.2.2 Multi-Level Checkpointing

In their 2010 paper [41], Moody, et al. make a case for multi-level checkpointing. They posit that not all failures are severe enough to justify restarting from the PFS and that other storage can also be used for checkpointing. They use multiple large systems for testing. There were three types of storage available: the PFS, node-local RAM, and Solid State Disks (SSDs). They plotted the aggregate bandwidth of checkpoint storage mechanisms as the size of the cluster grows, as shown in Figure 4.1.

The fastest storage in the figure is local RAM, but it is a suboptimal location for checkpoint storage; there are many failure modes which would make the RAM for one or more nodes unreachable. Pairing nodes together to form partners for

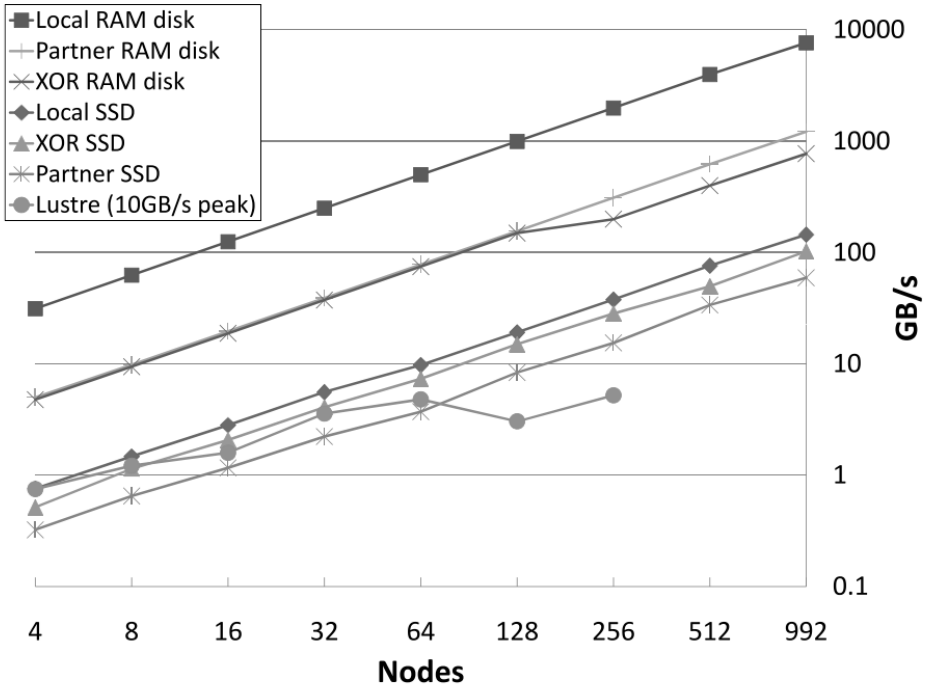

Figure 4.1: Aggregate write bandwidth based on storage technology [41]

checkpoint storage can increase the availability of the checkpoint. Partner nodes should be chosen to minimize the possibility of simultaneous failure. Note that the bandwidth of distributed storage in the cluster, including RAM and SSDs, scales linearly as the number of nodes increases; however, the PFS scales linearly only until it nears its theoretical maximum bandwidth.

The goal of multi-level checkpointing is to create a hierarchy for checkpoints which is similar to a memory hierarchy. The hierarchy creates the illusion of a large, fast, and reliable checkpoint storage. The frequency of checkpoints can then be increased beyond what the PFS can support. Only a subset of the checkpoints are stored to the PFS.

In the booster portion of the DEEP architecture, the amount of node-local RAM is limited. This increases the cost of using local RAM for checkpoint storage. Adding another level of storage is a possible solution to add capacity to the checkpointing hierarchy.

4.3 Design Space Exploration

Two important attributes of a new level of storage in the hierarchy are the interconnect and the storage technology. They dictate the price and performance of the device and how it can be used. This section presents some of the options for each attribute and discusses how the device could be used for checkpointing and generalized I/O.

4.3.1 Interconnect

The way in which the device is connected to the system is important because it determines the bandwidth, access granularity, and the possible side effects which the device can have on other elements of the system. Figure 4.2 shows some of the possible connection points present in the block diagram of a generic node: the memory interface, the I/O interface, the storage interface, and the network interface.

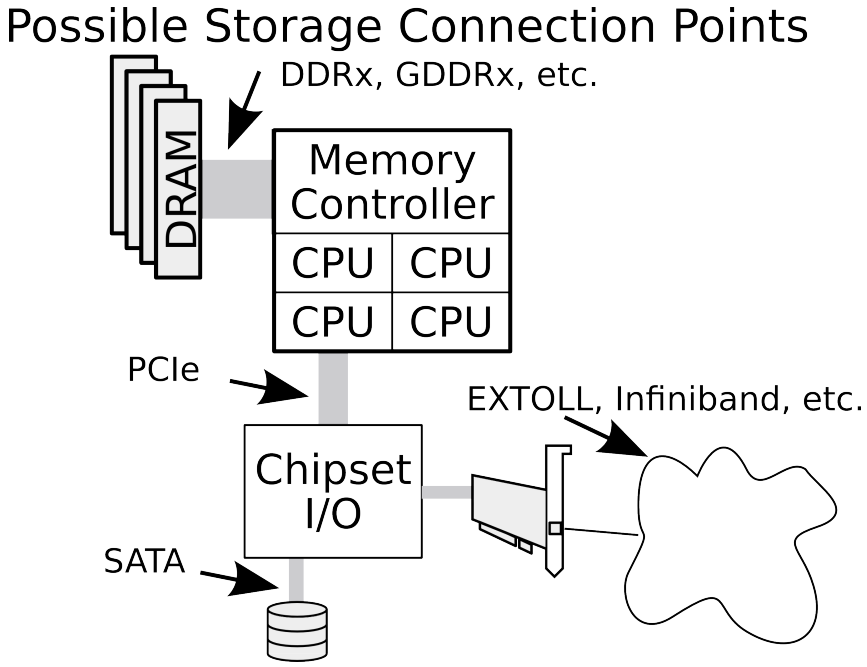

Figure 4.2: The block diagram for a generic node showing possible connection points for an extra level of storage.

The memory bus is the most attractive interface because it provides the highest bandwidth to the processor. As mentioned in chapter 3, the memory interface imposes strict latency requirements on devices. Memory interfaces are also relatively expensive, and in general it would be better to add more memory and partition it via software. In the case of the DEEP booster, simply adding more memory is not an option.

The I/O interface is attractive because it is designed to be a generic interface: constraints on latency are relaxed, all transactions are packetized, and it provides the next highest bandwidth connection. In the DEEP booster, the PCIe interface is directly connected to an EXTOLL network interface.

A storage interface such as SATA could also be used. It provides a narrow, block oriented interface to storage. The biggest downside of adding storage to the SATA interface is that it provides limited bandwidth. It is also block oriented, so fine-granularity transfers are not natively supported. In the DEEP booster architecture, the storage interface was left out to increase density and reliability.

The last interface considered here is the network interface. The biggest advantage of connecting to the network is that it separates the data from the node on which it was produced. Although other interfaces may provide higher bandwidth, in order to provide protection on a partner node, the data must travel over the network connection. The biggest disadvantage of using the network interface is that network interfaces provide different primitive operations, and the effort needed to implement a storage layer in the network varies accordingly.

4.3.2 Storage Technology

The storage technology used for this level of hierarchy is also important. Chapter 2 provides a more detailed description of possible storage technologies. For this discussion, it is important to recognize that DRAM, battery-backed DRAM, flash, or a byte-addressable NVRAM are all possible candidates.

DRAM provides the highest bandwidth and lowest latency, but is a volatile storage. This means that there are some failure scenarios where the stored data would be lost. This could be mitigated through the addition of a battery or capacitor, which would maintain the stored data when power is removed. Both of these solutions require a relatively large amount of power, even when not in use.

Using flash or a byte-addressable NVM would require less power, especially when not in use. Flash sacrifices byte addressability, but is readily available and relatively inexpensive. A byte-addressable NVM could provide a high-performance, low-power solution, but it is unclear when competitive NVM solutions will be available with sufficient density.

4.3.3 Checkpointing vs Generalized I/O Use Cases

Although the motivation for developing this added storage hierarchy level is to improve checkpointing reliability and performance, it can also be useful for accelerating general I/O. Checkpoint data in general is only collected for the case that a failure occurs. When no failure occurs, the checkpoint data can be erased to free storage. In general, I/O is the output of the application and should be preserved in any case. When a system supports application-level checkpointing, the line between I/O and checkpoints may be blurred. Some applications in the DEEP and DEEP-ER projects use the same format for checkpoints and I/O since they are based on iterative computations and write checkpoints and results at specified points in time.

4.3.4 Related Checkpointing Work

Other researchers have also seen an opportunity to add storage to the hierarchy. Checkpointing to node-local NVM can provide reasonable performance when combined with a pre-copy methodology. This reduces the amount of time that the processor must wait for the NVM, and the amount of bandwidth needed for remote checkpoints [42]. Some researchers have also recently proposed that replacing the RAM with NVM makes checkpointing unnecessary, but require a new type of non-volatile, transaction-based processes [43]. Since the promise of an NVM that will be competitive with DRAM is not new, there is also a large body of prior work in this area.

4.4 NAM Prototype

One of the purposes of the DEEP-ER project is to facilitate cross-disciplinary cooperation. In order to allow the NAM concept to be evaluated by software and reliability teams, building a functional prototype is included as part of the project. This section begins by describing the design space of such a prototype, starting with its essential functions and the choice of how to implement each function. It then describes how to include redundancy in the system and ends with a description of a modular NAM prototype.

4.4.1 Essential Functions

A minimal implementation of a NAM prototype contains an implementation of a network link, a storage controller, and storage. Each of these components could be implemented in FPGAs or ASICs. The normal tradeoffs for ASICs vs. FPGAs apply: ASICs have higher clock rates and higher density. This translates to higher performance for a specific power and area budget and lower cost in high volumes. However, ASICs are inflexible and expensive in small quantities. This means that for a prototype board, FPGAs will generally be preferred over ASICs when the logic may change or when additional features need to be implemented, but ASICs can reduce debugging effort and increase overall performance.

Useful features beyond those outlined above include: a DMA engine for large data transfers, a programmable storage controller for timing emulation of other storage technologies, the ability to connect the NAM to multiple network links for redundancy, and a PCIe link for status and testing. These features are not important enough by themselves to justify adding extra components to the NAM prototype, but will be implemented when the necessary resources are provided by the included components.

4.4.2 Choice of a Network

The DEEP-ER project was designed to provide flexibility in the choice of an interconnection network. With this idea in mind, the minimum specification for the network is that it must be compatible with a 16-lane PCIe slot. Infiniband, EXTOLL, and PCIe are all possible candidates.

Infiniband is an interconnection network which is sourced by multiple vendors including Mellanox. It has the largest share of Top500 systems with 222 systems on the June 2014 list. Infiniband networks consist of adapters for each node and high-radix switches, which are used to implement fat trees and other similar network topologies. One advantage of using Infiniband is that the NAM could be inserted in more locations in the network topology. Another advantage is the number of machines where the NAM could be inserted for testing. A disadvantage is the amount of effort needed to implement the Infiniband protocol. In order to be completed in a reasonable amount of time, the NAM would connect to the Infiniband adapter through PCIe, which would affect its latency and throughput.

EXTOLL is an interconnection network developed by the University of Heidelberg and a spin-off company, EXTOLL Gmbh. It is a switchless architecture, meaning

that the switching functions are incorporated into the network adapters. It is used to construct direct networks such as grids and tori. The switchless architecture means that the network cost scales linearly with the size of the network, which is an advantage of choosing EXTOLL. Another advantage of EXTOLL is the fact that it was also developed at the University of Heidelberg. This provides easy access to expertise for design, integration, and troubleshooting.

PCIe is a system-level interconnect which provides address-based routing and is based on reads and writes within the address space. It is being extended to provide more networking functionality and scalability. An advantage of PCIe is that it is a well understood standard. One disadvantage is that the networking extensions are not part of that standard. Another disadvantage is that its scalability has not been proven.

Until the DEEP-ER project chooses an interconnect technology, the decision for the NAM prototype can not be final. EXTOLL and PCIe will both be implemented in an FPGA for the prototype, and if Infiniband is chosen it will connect to the NAM through PCIe. It is vital that the FPGA is large enough to implement EXTOLL or PCIe along with the storage controller.

4.4.3 Storage Technology

The requirements for the storage in NAM prototype quickly narrow the list of candidate storage technologies. It must be available now, be byte addressable, provide at least a gigabyte of storage, and be able to match the bandwidth of the network. DRAM ends up being the most viable choice since flash is not byte addressable, and the density of SRAM is too low to make it a reasonable choice. Other emerging technologies such as PCM and FRAM currently have the same density problem as SRAM. A benefit of using DRAM for storage in the NAM prototype is that it is relatively inexpensive and is available with many different interfaces.

4.4.4 Storage Interface

Once the choice has been made to use DRAM for the NAM prototype, an interface must be chosen. The interface must be able to be implemented in an FPGA. The Hybrid Memory Cube interface was chosen because it provides sufficient storage and bandwidth and abstracts away the details of the RAM itself. This means that the controller implemented in the FPGA does not need to include support for ECC or refresh. The HMC memory controller is described in more detail in chapter 5.

4.4.5 Modular Design

Building a prototype with an unspecified network interface and a new memory technology carries some risk. In this work, the HMC board is designed as a separate board without an FPGA in order to increase the probability of a functional system. This allows the HMC interface to be tested with a reference board for the FPGA.

Figure 4.3 shows the four links of the HMC connected to various connector types on the HMC test board. Although an 8-lane link provides sufficient bandwidth to match a network link, 16-lane links are provided for future work. The connectors available are the FMC connector, an HDI-6 connector, and a PCIe edge connector.

The FMC connector implements a standard which is used on Xilinx evaluation platforms and is designed for daughter boards. It includes ten high-speed SERDES channels with a clock, an I^2C interface, and various general purpose I/O pins. Eight of the ten SERDES channels are used to provide an 8-lane HMC link.

The Samsung HDI-6 connector is the same connector used for EXTOLL. Each connector provides 12 SERDES channels, 9 of which are used to provide an 8-lane link with a clock. Two HDI-6 connectors can be used together to form a 16-lane HMC link.

The PCIe edge connector is used to provide a 16-lane HMC link. The rationale behind this decision is that there are many FPGA boards with a PCIe interface. A simple backplane board with a clock generator that allows two PCIe boards to communicate directly with each other was developed for the DEEP project. Using this backplane with an FPGA board provides an 8-lane link between the HMC and the FPGA.

Figure 4.4 illustrates how the HMC links from Figure 4.3 can be used to connect the HMC test board to the FPGA controller board or other HMC test boards. These connections are comprised of the HMC lanes and a clock signal since the HMC interface requires a synchronous clock. Sideband signals such as I^2C for configuration and controlling the reset signals must be connected separately.

In order to provide a synchronous clock regardless of the link connections, the HMC test board was designed to provide flexible clock distribution. A four-to-one multiplexer selects between the clocks provided by the MMCX connector, the HDI-6 connectors, or the PCIe link. A two-to-eight clock buffer then selects whether the clock should be taken from the output of the multiplexer or the clock which is generated from the on-board crystal oscillator. This clock is then driven to the HDI-6 links, to the FMC connector, and to the HMC.

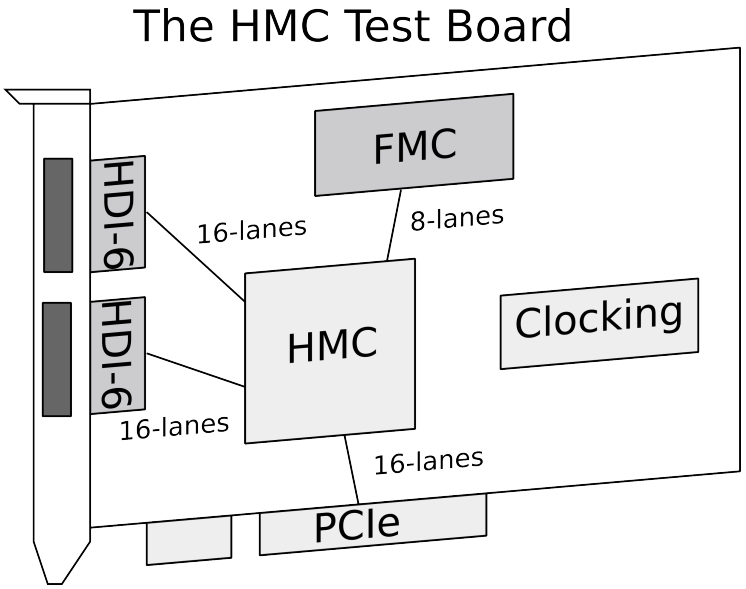

Figure 4.3: The block diagram of the HMC test board, showing the HMC link connections with their respective width in lanes.

4.4.6 Possible Topologies

The clock distribution circuitry allows all four links to be utilized at the same time, if the clock is generated by the PCIe backplane. Figure 4.6 illustrates some of the topologies that can be constructed using the HMC test board as a building block. Multiple controllers can be connected to the topologies to provide more bandwidth, as illustrated by the dashed connections in the mesh topologies. All topologies are limited to eight nodes since that is the maximum number of unique cube IDs supported by the HMC.

4.5 Related Work

Moneta [44] uses a PCIe-based board to model the performance of future memory technologies. The board consists of four Virtex 5 FPGAs, each with its own DDR2 interface. The FPGAs model a PCM controller, including access times and wear-leveling algorithms. The NAM connects to the network instead of to the PCIe interface and uses an HMC to provide the necessary memory bandwidth.

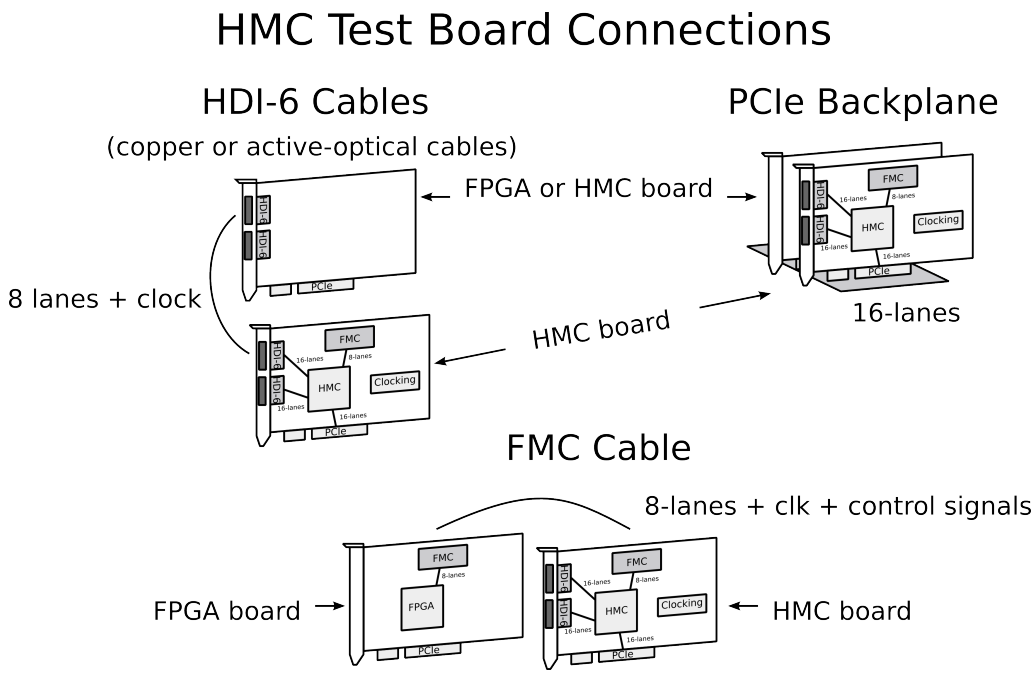

Figure 4.4: The connection possibilities for HMC test boards and FPGA controller boards.

4.6 Future Applications

Although the NAM is designed to be used as a memory extension for I/O and checkpointing applications, it could be used to accelerate other operations as well. Some of the features that could be modeled are atomics, global sums, and active memory.

Some atomic operations are already included in the HMC, both of which are immediate add instructions. The FPGA controller can be used to implement further atomic operations since the order of operations from a single memory controller to the HMC DRAM is guaranteed. Immediate operations such as compare and swap or compare and add can be implemented in a read-modify-write fashion. Atomic operations ease the development of parallel algorithms by avoiding certain classes of race conditions.

Another area of parallel processing that can be accelerated by NAM is global sums, or other global operations. Global calculations cost much more in terms of communication than calculation. For some applications, offloading the calculation to a NAM device could lower the communication overhead.

Active memory is a general term for operations that can be performed by the memory. Zeroing pages of memory, extracting features from matrices, and data

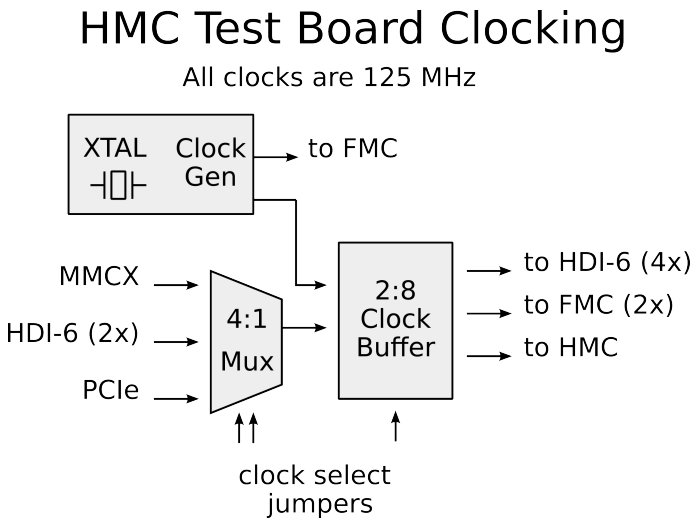

Figure 4.5: The clock distribution circuitry of the HMC test board, showing which clock sources can be used to drive the HMC and link clocks.

filtering are examples of possible applications for active memory. Active memory operations are similar to global operations because the cost of data movement is much larger than the cost of the calculations. The difference is that the data movement is not necessarily distributed throughout the cluster.

The combination of a network interface, high speed memory, and programmable logic provide a flexible prototyping environment for evaluating other potential uses of the NAM.

4.7 Conclusion

This chapter described the motivation for building the NAM, the design decisions involved, and the first part of the modular prototype. It is designed to connect with various FPGA boards and to provide flexibility in terms of possible topologies and clock distribution. Although the NAM is primarily targeted to increase I/O and checkpointing performance, it can also be used to model future non-volatile memory technologies and to provide atomic operations, global sums, and active memory functionality.

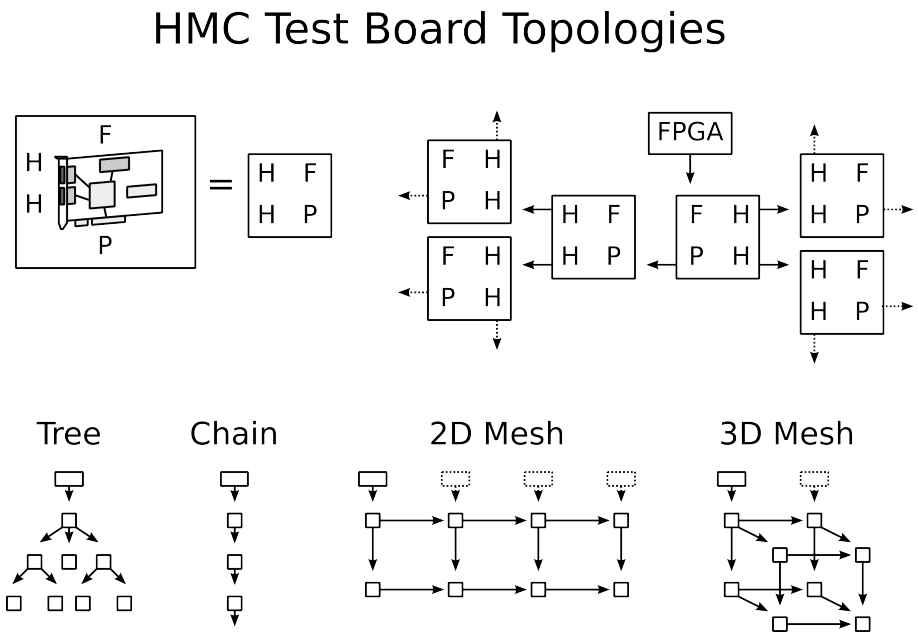

Figure 4.6: Some of the possible HMC topologies which can be constructed using the HMC test board.

5 Abstract Memory Interfaces

This chapter begins with a discussion of the historical trend toward narrow interfaces in computer systems and reasons why the interface to main memory is still parallel. Next, it discusses developments that may change the status quo, including the development of the Hybrid Memory Cube (HMC) specification. The details of the HMC interface and an implementation of a UVM-based test architecture are presented. This is followed by a discussion of the difficulties of prototyping the HMC in hardware, and a scalable packet processing architecture is presented as a partial solution to this problem. The chapter ends with a discussion of future narrow memory interfaces and whether memory interfaces and network interfaces will converge.

5.1 Narrow, High-Speed Interfaces

One of the basic characteristics of a physical interface is its width. The width of an interface is also called a phit (physical bit), or the number of bits of data which can be transferred in a single bit time. As the architecture for personal computers has evolved over the decades, wide interfaces have been replaced by narrower ones: PCI, PCI-X, and AGP have been replaced by PCIe; PATA (formerly ATA) has been replaced by SATA; and the front-side bus architectures from Intel and AMD have been replaced by QPI and HyperTransport™, respectively. Some of the forces driving the adoption of narrow interfaces include density and cost, efficiency, and even cooling [45].

The number of pins which can be included on a package has not scaled with the number of transistors. This disparity has increased the pressure on designers to increase bandwidth while minimizing the increase in pincount. Increasing the frequency of the interface allows the designer to increase bandwidth, even while decreasing its width.

Narrow interfaces are denser in terms of connectors, cabling, and board space. As long as the pitch of the connector pins or the size of the conductors in a cable is

held constant, increasing the width of the interface will increase the size and cost of the connectors and cables. The effect on board space is similar, but it may be worse because the routing complexity can increase superlinearly with interface width.

Narrow interfaces may decrease the penalty for unused bandwidth. The efficiency of wide interfaces suffers when the transferred data is not a multiple of the phit size. Although there is no more data to be transferred, partial phits can not be transferred. On a narrow interface, the next transfer is able to start sooner when less data is transferred, which reduces the amount of unusable bandwidth.

In the case of the PATA to SATA conversion, the width of the PATA ribbon cable was a significant fraction of the width of the machine, and was known to interfere with the cooling airflow.

In a modern personal computer, there are very few parallel connections left to be serialized.

5.1.1 The Memory Exception

One of the last parallel interfaces in a modern computer system is the interface to main memory. Several attempts have been made to design a popular narrow memory interface, but none of them has been as widely deployed as the parallel interfaces. One of the important reasons for this has been a focus on single-threaded performance, which is very dependent on memory latency. Although high-speed links can increase the maximum bandwidth of the interface, serialization and deserialization add latency to each transfer, as illustrated in Figure 5.1. Extra cycles spent waiting for data from memory translate directly to higher application latency, or reduced performance. Another source of overhead with high-speed serial links is the need to add methods for error detection, such as Cyclic Redundancy Code (CRC) calculations since the bit-error rate increases with bit rate. As discussed in Chapter 2, there can also be other drawbacks to narrow memory interfaces, such as increased power consumption and cost.

5.2 Independent Banks and Controllers

Recent developments in computer architecture and the continuing advancement of process technology indicate the need for a change. Increasing the number of memory channels and banks may be sufficient to maintain system performance as long as the amount of thread-level parallelism, i.e. the number of independent memory request

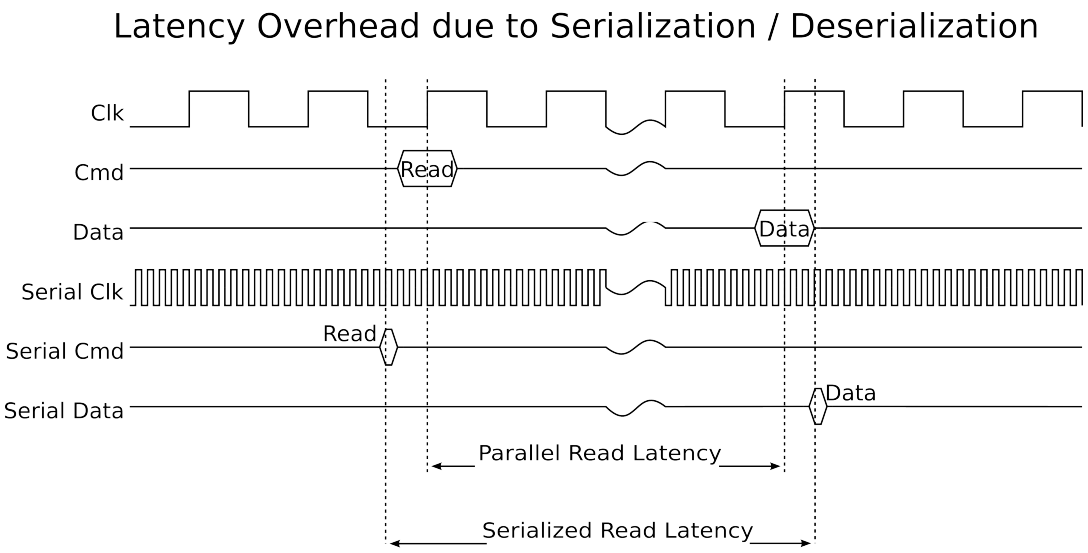

Figure 5.1: An illustration of the added latency due to serialization.

streams, is low. This is an expensive option in terms of package pins and board space, but it allows for the use of commodity memory parts, and a premium is charged for each additional memory slot and interface. As the frequency of the interface increases, the number of memory banks which can share a channel decreases, due to capacitative loading and increased trace length. DDR4 is defined as a point-to-point interface, as opposed to a multi-drop interface, meaning that only one memory module can be connected to a DDR4 interface. This allowed the frequency to be increased further, at the cost of reducing the number of memory banks per channel.

5.2.1 Many Core Emphasizes Average Latency

For many years, processing performance scaled with the clock frequency. Chip manufacturers have largely stopped scaling frequency, however, because they have reached the end of Dennard scaling [46]. Processor manufacturers have, therefore, increased the number of processing cores per chip in order to improve performance. This shifts the burden of increasing performance to software developers who are tasked with finding and exploiting thread-level parallelism. As mentioned previously, this also increases the number of independent memory request streams. This is especially evident in the GPGPU space, where GPU manufacturers have added features to their graphics engines to support general computations.

Independent of the processor type, when the number of memory request streams becomes larger than the number of memory interfaces, the probability of waiting increases because the interface is busy with other requests. In this case, the average

latency of memory accesses becomes more important than the minimum latency. This affects the memory system architecture as well as the interface of the RAM. For example, GDDR has been the memory of choice for graphics cards instead of the less expensive DDR since it provides higher bandwidth access to memory.

5.2.2 Power Considerations

DRAM architectures contain row buffers to improve average read latency and refresh times, as Section 2.2.2 explains. Mixing multiple independent request streams effectively decreases the locality of the request stream. This reduced locality translates to wasted power in large row buffers [4], [47] since the data is not used before the row buffer is reused. This can be mitigated through scheduling, which separates streams and recovers data locality, but this increases the power consumption and complexity of the architecture.

5.2.3 Commercial Risk

As with any standard, backing the wrong memory standard can be very costly. The market favors incremental, backwards compatible changes. Incremental advances are easier to implement, easier for the consumer to understand, and therefore minimize risk. When the memory interface changes, the memory controller must be replaced. When the memory controller was part of a separate chip, this could be accomplished without re-designing the processor. As long as the front-side bus was compatible, memory controllers and interfaces could be updated independently from the processor. Recently, integrated memory controllers have removed this possibility.

An integrated memory controller is advantageous because it causes the bandwidth to memory to scale with the number of processors, decreases the latency of memory accesses, and decreases the component count. Some of the drawbacks of integrated memory controllers are the synchronization of processor and memory interface development cycles, the increased pin count of the processor package, in addition to the increased commercial risk.

5.3 Storage Abstraction

Memory controllers are technology specific. When processors are designed with an integrated memory controller for a specific technology and interface, there is very little flexibility. Voltage levels, command set, and the timing parameters are

dictated by the type of memory supported. The values of timing parameters such as the Row-Address Strobe (RAS) and Column-Address Strobe (CAS) delays can be configured and are generally included in a Serial Presence Detect ROM (SPD) on the memory module. Memory controllers which are implemented in FPGAs are even less flexible. In order to save logic resources and improve timing, they are often compiled for specific RAM modules.

There are a number of benefits which could come from making the connection between the processor and the RAM more flexible, which are described in the following sections. One cost of these benefits would be the additional latency introduced by the communication overhead.

5.3.1 Storage Technology Agnostic Interface

As noted in Chapter 2, there are many competing memory technologies with various advantages and disadvantages in terms of price, scaling, volatility, etc. If each of these memory technologies were implemented with the same interface, one could imagine building a range of systems using the same processor to target different market segments or applications. Another benefit would be the possibility of building hybrid systems where the software controlled what was stored in different types of memory.

5.3.2 Repartitioned Controller Functions

Removing the memory controller from the processor would return some of the flexibility lost to integration. Memory controllers could then be connected to the processor via narrow, high-speed links [48]. The downsides would be increased component count and increased power consumption. Figure 5.2 shows the progression of the architecture of a system as the memory controller and more processors are integrated into the same package in A, B, and C. D, E, and F illustrate some of the possible directions in which system architecture could progress: package-level integration, the buffer-on-board [49] architecture, and integrating the memory controller and the memory in the same package. Package-level integration further reduces the flexibility of the interface, while the buffer-on-board architecture and the memory and controller integration increase the flexibility of the architecture.

Package-level integration brings the RAM into the same package with the CPUs and memory controllers. Memory interfaces such as WideIO and High-Bandwidth DRAM are based on the availability of the high-density interconnection possibilities

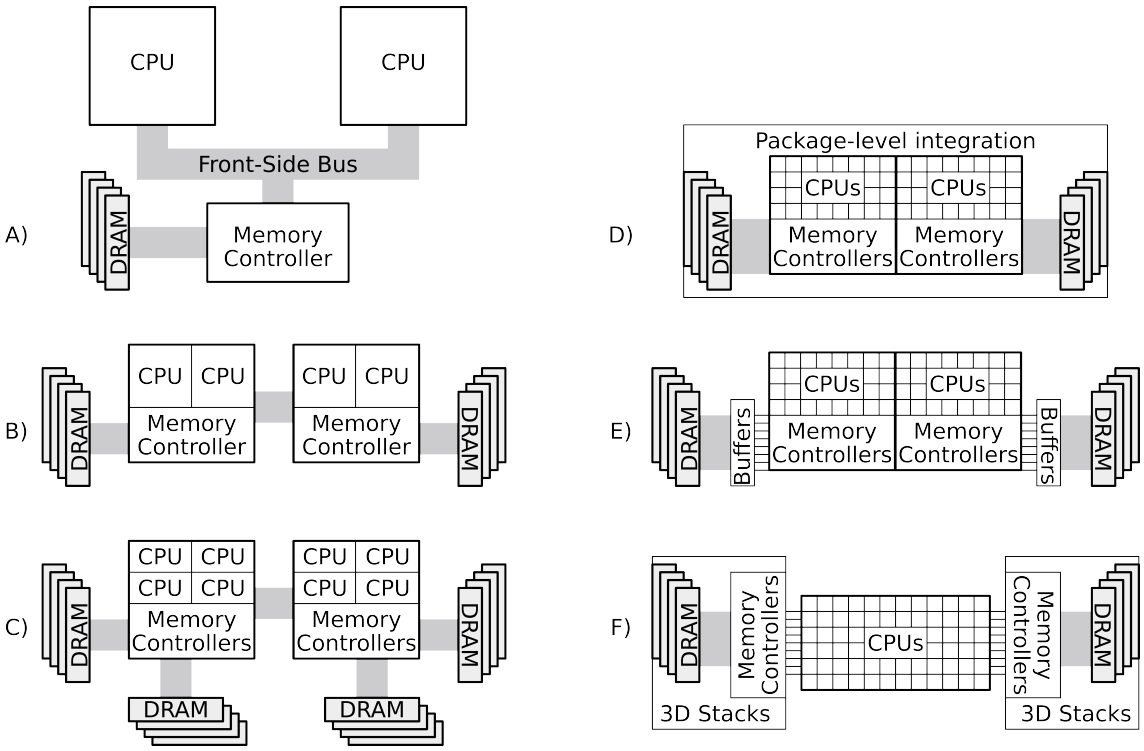

Figure 5.2: DRAM attachment to CPUs and memory controllers. A, B, and C show the progressive integration of components, and its effect on the memory hierarchy. D, E, and F show some of the architectures which have been proposed.

which this provides. The proximity of the CPUs and the memory should reduce latency and the power consumption of the interface. One of the problems with this approach is its inflexibility in terms of the ability to upgrade and replace the RAM.

Buffer-on-board replaces the interface from the memory controller to the memory with a high-speed narrow interface to a separate chip. This increases the flexibility of the architecture since the number of memory interfaces is separated from the number of processor interfaces. This also allows other memory topologies to be implemented. For example, a tree of buffers could be built to increase the total memory capacity for a given number of interfaces. A downside of this approach is the need for another component, which increases system cost, power consumption, and memory latency.

Separating memory controllers from the processors increases the flexibility available to memory system designers. This flexibility is desirable in order to take advantage of emerging memory technologies, and the possibility of having multiple memory technologies in the same memory hierarchy. This comes at the cost of an additional interface, which increases power consumption and latency. One way to avoid this extra cost is to remove the chip-to-chip interface from the memory controller to

the memory, i.e., integrate the controller with the memory. This maintains the advantages of separate memory controllers and increases the possible advantage of local refresh.

5.3.3 Scalable Performance

Having a narrow interface from the processor to the memory controller allows the processor to include more memory interfaces with the same number of pins or to reduce cost by decreasing the size of the package. It also increases the number of possible ways to connect the memory controllers since bandwidth and capacity can be independently scaled. Bandwidth can be scaled by adding memory interfaces, while capacity is scaled by adding storage to the memory controller.

5.3.4 Local Refresh Functionality

There is an extra consideration for volatile memory, which needs to be regularly refreshed to avoid data loss. Refreshing the RAM consumes a significant amount of energy due to the chip-to-chip communication required to read the stored values and re-write them. Controlling it locally creates energy savings in two ways: through finer granularity of refresh control and reduced communication overhead.

5.4 Hybrid Memory Cube

The HMC is an abstract memory interface developed by the HMC Consortium [50]. It was designed for a stacked DRAM architecture, but could be used with other storage technologies. It is the first memory interface which resembles a direct network with integrated switching. This allows it to be more flexible in terms of the types of memory which are combined and the memory topologies which can be created. The first commercially available implementation integrates a memory controller with stacked DRAM. This section discusses the design and features of the HMC architecture, its protocol, and some of its advantages and disadvantages.

5.4.1 Tuned Logic and Storage Processes

The HMC protocol is designed to connect processors with stacked DRAM [51]. DRAM manufacturers plan on stacking DRAM to increase storage density, tune the

DRAM process independent of the logic process, and maintain yield via redundant structures.

As technology scales and frequency increases, on-die signaling becomes a larger problem. This is especially true for storage technologies where large, regular arrays of storage elements must be connected to sense amplifiers and output pins. One way to reduce the total distance that these signals must travel is through die stacking using Through-Silicon Via (TSV) technology. TSVs are analogous to vias in printed circuit boards. There are multiple varieties of TSVs, but they are all based on thinning processed wafers, boring holes through them, and building wafer-to-wafer connections through the holes [52]. Stacking multiple wafers means that each layer can be different in terms of layout or fabrication process.

The HMC design takes advantage of this opportunity to specialize the wafer stacks by separating the logic and DRAM storage functions. DRAM and logic are traditionally fabricated with different process technologies because the transistors are optimized for different characteristics. Logic transistors are tuned to provide fast operation, which means that they must have a high switching current. DRAM processes are tuned for low leakage to allow the DRAM cells to maintain their charge for as long as possible. Both logic included in a DRAM process as well as DRAM fabricated on a logic process are suboptimal. Separating the functions overcomes this problem, but increases the cost of manufacturing since a fault in one die can render the stack nonfunctional.

Including redundant TSVs, DRAM cells, and routing resources mitigates the risk of decreasing yields. The redundant resources can be managed by the logic layer, and may also be used for repairing faults which occur after the device has been deployed. The logic layer can also change address mappings in order to avoid hard faults. The amount of redundancy and fault tolerance can be traded against yield and cost.

5.4.2 The HMC Short Range Interface

There are two interfaces for the HMC, the short-range interface and the ultra-short-range interface [50]. The ultra-short-range interface has not been fully specified, and current HMC silicon uses the short-range interface, which is described in this section. The description begins with the pins and routing constraints and then describes the packet format and serialization.

5.4.2.1 Pins

An HMC link consists of 16 differential pairs and a power state pin in each direction, as shown in Figure 5.3 and Table 5.1. These pins are necessary for each link and are prefixed with the link number: L0, L1, etc. In addition, there is a differential reference clock, a global reset, and a pin to indicate fatal errors. These pins must be present, but are shared among all the links of a device. The last group of pins includes pins that select the frequency for the reference clock, the termination mode for the reference clock, and the ID for the HMC. These configuration pins allow the PLL to lock automatically, and sets the address for the I^2C bus. This group is separate in the table because this functionality could be moved to configuration registers or encoded differently without affecting the operation of the interface. Current HMC devices have an I^2C and a JTAG interface for configuration and debugging. These pins are not part of the interface, could be implemented in another way, and are not included in the table.

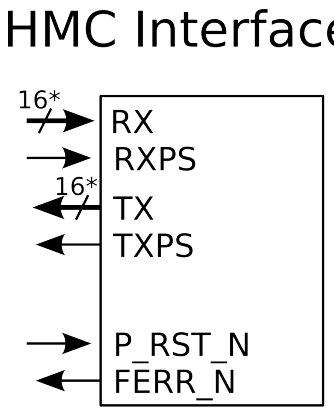

Figure 5.3: The block diagram of the HMC short-range interface, showing the serial lanes (TX and RX), the power state pins (TXPS and RXPS), and the reset (PRST_N) and fatal error (FERR_N) pins. The interface is also presented in Table 5.1 .

5.4.2.2 PCB Routing Simplification

One of the stated goals of the HMC specification is to simplify routing and reduce the amount of board space needed for a memory interface. This goal is furthered by the inclusion of lane reversal and lane polarity inversion support. Lane reversal means that the transmitting lanes can be connected to the receiver in reverse order. The lane connection order is determined during link initialization and corrected

Signal pins for each link "x", prefixed with "Lx"

Name	Type	Description
LxTXP[15:0] and LxTXN[15:0]	output	Transmitting lanes (differential pairs)
LxTXPS	output	Transmitter active power state
LxRXP[15:0] and LxRXN[15:0]	input	Receiving lanes (differential pairs)
LxRXPS	input	Remote Transmitter active power state

Signal pins for each cube

Name	Type	Description
REFCLKP and REFCLKN	input	Reference clock (differential pair)
P_RST_N	input	Reset (asserted low)
FERR_N	output	Fatal Error

Config pins for each cube

Name	Type	Description
REFCLKSEL	input	On-die termination selection for REFCLK
REFCLKBOOT[1:0]	input	Frequency selection for REFCLK
CUB[2:0]	input	Cube ID selection

Table 5.1: The pins of the HMC short-range interface can be separated into three groups.

internally, if necessary. Lane polarity inversion means that for each differential pair, the connections to the receiver can be made in two ways: TXN to RXN and TXP to RXP or TXN to RXP and TXP to RXN. The polarity of the signal is inverted, if necessary, within the chip. Simplified examples of lane reversal and lane polarity inversion are shown in Figure 5.4.

5.4.2.3 Packet Types

All information on the HMC link is transmitted via request, response, or flow packets. This reduces the number of pins necessary to implement the interface and enables multiple outstanding requests since the HMC interface implements split-phase transactions for responses. Multiple outstanding requests and out-of-order read responses increase the amount of parallelism available through HMC interface. The HMC guarantees that requests to the same memory location from the same host link will be performed in order, but further ordering must be handled by the processor.

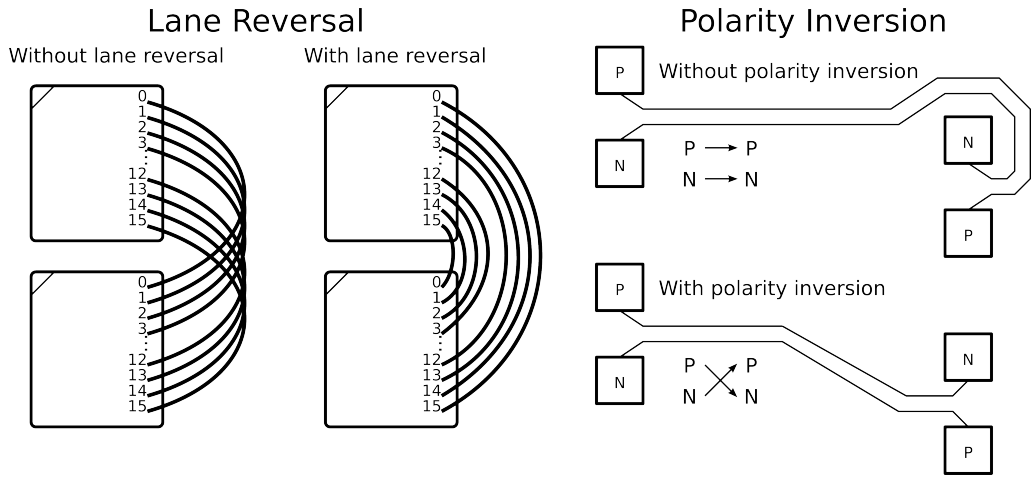

Figure 5.4: In order to simplify routing, the HMC interface supports lane reversal and lane polarity inversion.

5.4.2.4 Routing and Extensibility

The HMC specification makes provision for the creation of small networks of HMCs. Each request packet contains a three-bit Cube ID (CUB), which specifies its destination, and a three-bit Source-Link ID (SLID), which specifies the host link from which it entered the network. When a response is generated, its SLID bits are set to the SLID from the request. Each HMC has an internal crossbar and routing tables to map CUB (requests) and SLID (responses) values to destination links. These routing tables are configured by the host through register reads and writes, either through special HMC packets or through the JTAG or I^2C interfaces.

5.4.2.5 Flow Control

Like any high-bandwidth network, the HMC interface needs flow control to ensure that no packets are lost. HMC devices store received packets in a buffer until they can be sent on to their destination or processed further. If more packets are received than there is space in the buffer, packets will be lost. In order to avoid this, HMC devices must keep track of the amount of buffer space, measured in flits, which is available. As part of initialization, a device sends Token RETurn (TRET) packets to initialize the count. From this point on, the number of tokens is decremented when packets are sent, and packets may only be sent when there are enough tokens. If tokens take too long to be returned or are not returned, they will limit the bandwidth of the link or cause it to cease sending packets. Request, response, and TRET packets can return tokens to the other side of the link. Flow packets are never sent

on and are not stored in the buffer. They do not consume tokens and can be sent even when there are no available tokens.

5.4.2.6 Packet Overhead

An HMC packet consists of one or more 128-bit flits. The first half flit of a packet to be transmitted is the header, the last half flit is the tail, with data coming between the header and the tail, as illustrated in Figure 5.5. A single packet can contain up to eight data flits, or 128 bytes of data. The header and tail contain routing information, address and control bits, and a CRC. This information is added to support the network functionality of the HMC and to increase its reliability at the cost of reducing the effective bandwidth of the link. These fields and features are discussed further in the following sections.

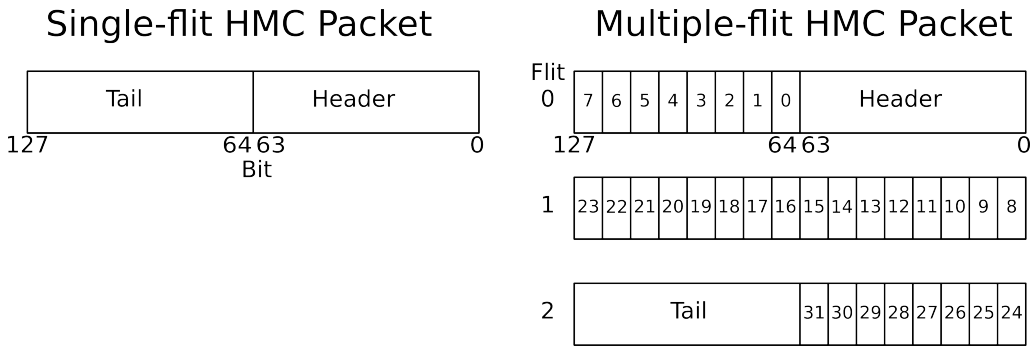

Figure 5.5: HMC packet layout examples.

5.4.3 Clocking

The HMC short-range interface is a synchronous interface; both ends of the link use a multiple of the same reference clock to send data. This simplifies the design of an HMC controller since there is no need to worry about buffer overflows caused by frequency differences between a transmitter and a receiver. Although the clocks are synchronous, the phase of the received data on each lane may be different, due to differences in trace lengths. Therefore, each lane contains Clock Data Recovery (CDR) circuitry to align the data with the sampling clock.

5.4.3.1 Run-Length Limitation

In order to maintain alignment, the CDR circuit needs to see transitions in the incoming data stream. The HMC specification limits the number of bit times without

a transition to 85. This means that a transmitter must force a transition after transmitting 85 consecutive zeros or ones.

5.4.3.2 Scrambling

Scrambling data before transmitting it increases the probability of having enough transitions on the link for the CDR circuit. The HMC scrambles the data by XORing the output data with the least significant bit of a Linear Feedback Shift Register (LFSR). An LFSR is a simple circuit which cycles through a set of values, each value being a function of its last value. The initial value is referred to as the seed.

Figure 5.6 shows the schematic of a simple three-bit LFSR, which implements the function $1+x^{-2}+x^{-3}$. It can be generalized to implement the function $1+x^{-(n-1)}+x^{-n}$ by adding $n-3$ more register stages between stage 1 and stage 2, which is represented with a dashed line. An LFSR of this form cycles through 2^n-1 values if it is initialized with a non-zero seed, and remains zero if the seed is zero. If the registers are initialized with the seed 001, the LFSR will cycle through the values shown in Table 5.2. The table also shows the values given when $n = 4$ and $n = 15$, which is used by the HMC.

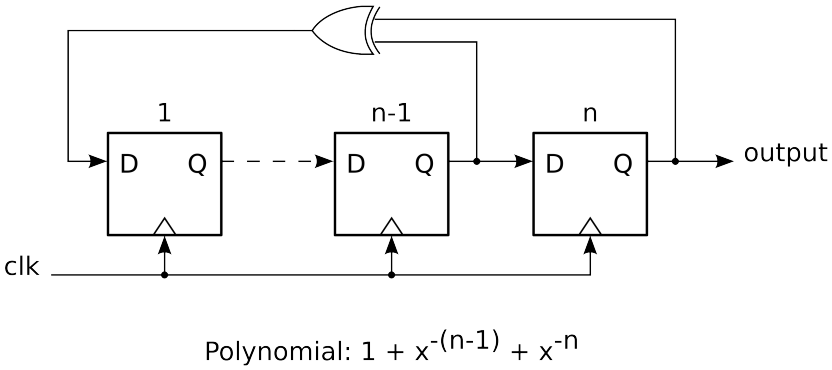

Figure 5.6: The schematic for a simple LFSR.

The value of the LSB transitions at least every n bits. This can be observed from the values for the LSB for each LFSR in Table 5.2 and the schematic diagram in Figure 5.6. This provides the bit transitions necessary for the CDR circuitry, except in two special cases: when the data to be transmitted matches the scrambler output or is the inverse of the scrambler output. In these cases, the output will be all zeros or all ones, respectively. The chance of this happening is approximately 2^{-80} for random payload data [50].

Count	**Expression**	$n = 3$	$n = 4$	$n = 15$
1		3'b001	4'b0001	15'b000000000000001
2		3'b100	4'b1000	15'b100000000000000
3		3'b010	4'b0100	15'b010000000000000
4		3'b101	4'b0010	15'b001000000000000
5		3'b110	4'b1001	15'b000100000000000
6		3'b111	4'b1100	15'b000010000000000
7	$(2^3 - 1)$	3'b011	4'b0110	15'b000001000000000
8		3'b001	4'b1011	15'b000000100000000
9		...	4'b0101	15'b000000010000000
10			4'b1010	15'b000000001000000
11			4'b1101	15'b000000000100000
12			4'b1110	15'b000000000010000
13			4'b1111	15'b000000000001000
14			4'b0111	15'b000000000000100
15	$(2^4 - 1)$		4'b0011	15'b000000000000010
16			4'b0001	15'b100000000000001
17			...	15'b110000000000000
18				15'b011000000000000
⋮				⋮
32766				15'b000000000000111
32767	$(2^{15} - 1)$			15'b000000000000011
32768				15'b000000000000001
32769				...

Table 5.2: Values for an LFSR of the form $1 + x^{-(n-1)} + x^{-n}$ with different values of n.

Lane	Seed	Sequence Number	Distance
0	15'h4D56	0	0
1	15'h47FF	1952	1952
2	15'h75B8	3872	1920
3	15'h1E18	5824	1952
4	15'h2E10	7728	1904
5	15'h3EB2	9648	1920
6	15'h4302	11584	1936
7	15'h1380	13504	1920
8	15'h3EB3	15424	1920
9	15'h2769	17376	1952
10	15'h4580	19280	1904
11	15'h5665	21200	1920
12	15'h6318	23120	1920
13	15'h6014	25056	1936
14	15'h077B	26976	1920
15	15'h261F	28896	1920

Table 5.3: Scrambler seeds per lane with their respective sequence numbers and the distance, in steps, between them.

The scrambler for each lane is specified to have a unique seed, in order to minimize the correlation between lanes. Table 5.3 lists the seed values, their sequence numbers when the seed for lane 0 is assigned the number 0, and the distances between successive seeds. These values could be used to calculate the effectiveness of the scrambling, and can be useful for testing the run-length limitation circuitry. It is interesting to note that the seeds are not equidistantly spaced throughout the possible space, and it would be interesting to know how the seeds were chosen.

It is important to recognize that the scrambler state is independent of the data which is transmitted, and that future seeds can be calculated with simple combinational logic. This allows parallel scrambler circuits to be implemented, which relaxes the timing constraints for the design [53].

In order to allow the receiver to descramble the data, the training sequence includes a phase where scrambled zeros are transmitted. These scrambled zeros transmit the LSB of the scrambler. After collecting n bits of scrambled zeros, the state of the LFSR $n-1$ bit times ago is known. The current seed can then be calculated and loaded into the LFSR. The received data can then be compared with the output of the LFSR to ensure that the descrambler is correctly seeded.

5.4.4 Error Prevention and Detection

As with any high-speed link, there is a non-negligible probability of bit errors. The HMC specification defines CRC coding, replicated length fields, and sequence numbers to detect bit errors and prevent link failures.

5.4.4.1 Cyclic Redundancy Codes

A CRC provides a method of detecting bit errors which occur during transmission. A CRC is formed by treating the contents of the message as a polynomial, and dividing it by the generator polynomial. Generator polynomials take the form shown in (5.1) and are referred to using CRC-n, where n is the highest power of x. Note that C_n and C_0 are not shown since they are equal to one by definition. Two common representations of CRC polynomials take advantage of this fact. The CRC-8 polynomial shown in (5.2) can be represented by the 9-bit hexadecimal value 0x131, as shown in (5.3), but it can also be represented with an implicit MSB as 0x31 (5.4), or with an implicit LSB as 0x98 (5.5). This work uses the representation with the implicit MSB.

$$x^n + C_{n-1} \cdot x^{n-1} + C_{n-2} \cdot x^{n-2} + ... + C_2 \cdot x^2 + C_1 \cdot x^1 + 1 \tag{5.1}$$

$$1 \cdot x^8 + 0 \cdot x^7 + 0 \cdot x^6 + 1 \cdot x^5 + 1 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0 \tag{5.2}$$

$$C_8C_7C_6C_5C_4C_3C_2C_1C_0 = 100110001(binary) = 0x131(hexadecimal) \tag{5.3}$$

$$C_7C_6C_5C_4C_3C_2C_1C_0 = 00110001(binary) = 0x31(hexadecimal) \tag{5.4}$$

$$C_8C_7C_6C_5C_4C_3C_2C_1 = 10011000(binary) = 0x98(hexadecimal) \tag{5.5}$$

Table 5.4 contains a simple example calculation for an 8-bit message with the CRC-8 polynomial used in the Dallas One-Wire bus [54]. In the first step, the message (8'b10101011) is padded with 16 bits of zeros, i.e., twice the length of the

Bit	Remainder	Padded Message	Operation
0	8'b00000000	24'b00000000_00000000_10101011	Zero and Pad
1	8'b00000001	24'b00000000_00000000_10101011	Shift
2	8'b00000011	24'b00000000_00000000_10101011	Shift
3	8'b00000110	24'b00000000_00000000_10101011	Shift
4	8'b00001101	24'b00000000_00000000_10101011	Shift
5	8'b00011010	24'b00000000_00000000_10101011	Shift
6	8'b00110101	24'b00000000_00000000_10101011	Shift
7	8'b01101010	24'b00000000_00000000_10101011	Shift
8	8'b11010101	24'b00000000_00000000_10101011	Shift
9	8'b10011011	24'b00000000_00000000_10101011	Shift and XOR
10	8'b00000110	24'b00000000_00000000_10101011	Shift and XOR
11	8'b00001100	24'b00000000_00000000_10101011	Shift
12	8'b00011000	24'b00000000_00000000_10101011	Shift
13	8'b00110000	24'b00000000_00000000_10101011	Shift
14	8'b01100000	24'b00000000_00000000_10101011	Shift
15	8'b11000000	24'b00000000_00000000_10101011	Shift
16	8'b10110001	24'b00000000_00000000_10101011	Shift and XOR
17	8'b01010011	24'b00000000_00000000_10101011	Shift and XOR
18	8'b10100110	24'b00000000_00000000_10101011	Shift
19	8'b01111101	24'b00000000_00000000_10101011	Shift and XOR
20	8'b11111010	24'b00000000_00000000_10101011	Shift
21	8'b11000101	24'b00000000_00000000_10101011	Shift and XOR
22	8'b10111011	24'b00000000_00000000_10101011	Shift and XOR
23	8'b01000111	24'b00000000_00000000_10101011	Shift and XOR
24	8'b10001110	24'b00000000_00000000_10101011	Shift

Table 5.4: An example CRC calculation using the 8-bit polynomial 0x31.

remainder, and the remainder is reset to zero. In each step, the bit from the padded message which is shifted into the remainder is highlighted. When the MSB of the remainder is set, the next shift will cause an overflow, and the new remainder is XORed with the polynomial 8'b00110001(0x31). After bit 24 has been shifted in, the remainder is 8'b10001110, the final CRC value. Note that changing one bit in the input results in multiple bits being changed in the calculation since each overflow bit causes 3 bits to change in the remainder on the next cycle, as shown by the highlighted bits in the remainder column. Since the calculation only consists of XOR and shift operations, the CRC can also be calculated in parallel.

The CRC calculation for the HMC is very similar. The initial remainder is set to zero, and the message is padded with 64 bits of zeros after the MSB. After the last step, the remainder is the CRC value, which is inserted into the packet or checked against the CRC value from a received packet.

The HMC specifies the use of the Koopman-CRC32K [55] polynomial shown in 5.6, and represented by the value 0x741B8CD7. For single-flit packets, this polynomial has a Hamming distance of eight, meaning that seven independent bit errors can be detected. For packets up to the HMC MTU of nine flits, it has a Hamming distance of six.

$$x^{32}+x^{30}+x^{29}+x^{28}+x^{26}+x^{20}+x^{19}+x^{17}+x^{16}+x^{15}+x^{11}+x^{10}+x^{7}+x^{6}+x^{4}+x^{2}+x^{1}+1 \tag{5.6}$$

The choice of the polynomial for the CRC calculation affects its effectiveness in detecting bit errors and the complexity of its calculations. For a hardware implementation, the number of non-zero coefficients of the polynomial determines the number of XOR gates which are needed. The polynomial used for the HMC has 18 non-zero coefficients. If the polynomial shown in 5.7 (0x20044009) were used, its 7 non-zero coefficients would significantly reduce the amount of logic needed to implement the calculation, but the Hamming distance of the CRC would be reduced to 6 for single-flit packets [55].

$$x^{32} + x^{29} + x^{18} + x^{14} + x^{3} + 1 \tag{5.7}$$

For a software implementation there is no difference in the number of operations, as long as a 32-bit XOR must be used. Special classes of CRC-32 polynomials have been found which confine the non-zero coefficients to lower bits. CRC32sub16 and CRC32sub8 are two of these classes. The polynomial in 5.8 (0x000000E5) is a member of of the CRC32sub8 class, which would also provide a Hamming distance of 6 for HMC packets [56]. A CRC calculation based on this polynomial could be implemented with an 8-bit XOR.

$$x^{32} + x^{7} + x^{6} + x^{5} + x^{2} + 1 \tag{5.8}$$

5.4.4.2 Replicated Length Fields

As previously shown in Figure 5.5, the CRC is contained in the tail of a packet. In order to check whether a packet was received correctly, the length of the packet must be known. HMC packets include two copies of the packet length as protection against bit errors in the length field. This reduces the probability that a corrupted packet is accepted since at least two bit errors must occur, one in the length field

and one in the duplicate length field.

As an alternative to duplicating the length field, the length could be protected by parity or otherwise encoded. Parity provides the same level of protection against bit errors, in terms of Hamming distance, at the cost of one bit instead of four. This would save a couple of bits in the header for other uses, such as providing more address bits.

5.4.4.3 Sequence Numbers

All request packets and some flow packets contain a 3-bit sequence number, which provides another check to make sure that data is not corrupted. Packets which are received out of order are discarded and initiate a retry sequence, as described in Section 5.4.5.

5.4.5 Retransmission

In order for the HMC link to provide a reliable connection, packets of most types must be stored in case they need to be sent again at some later point. Missing writes would cause inconsistency in the data, missing reads or responses could cause a memory request stream to stall waiting for data, and missing TRET packets could degrade performance, eventually causing traffic on the link to cease, as discussed in Section 5.4.2.5.

When a failure is detected, the HMC retransmits packets starting from the packet after the last packet which was correctly received. This is implemented using a retry buffer, Forward Retry Pointers (FRP), and Return Retry Pointers (RRP).

5.4.5.1 Retry Buffer

Requests, responses, and TRETs which are sent across the HMC link are stored in the retry buffer. The retry buffer, shown in Figure 5.7, is organized into a circular queue, with a read pointer and a write pointer. It can be sized independently from the input buffers since it must only hold packets until they are acknowledged. The retry buffer can hold up to 256 flits since the read and write pointers are 8-bit values.

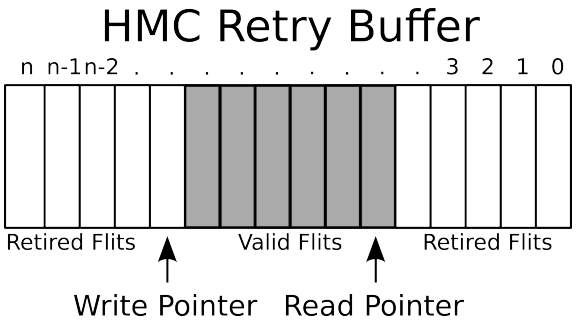

Figure 5.7: The retry buffer holds the flits of transmitted packets until they have been acknowledged.

5.4.5.2 Forward Retry Pointers

Every packet which is stored is assigned a Forward Retry Pointer (FRP) before it is transmitted. The FRP points to the entry in the retry buffer where the header flit of the next packet will be stored, which is the updated value of the write pointer. This value is inserted in the tail of the packet before it is stored in the retry buffer. When the packet is successfully received, this gives the correct value for where to start a possible retry.

5.4.5.3 Return Retry Pointers

The read pointer of the retry buffer is updated when a valid packet is received with an Return Retry Pointer (RRP) larger than the current read pointer. This allows the buffer space to be reused for new packets. Since the RRP is the FRP of the last correctly received packet, the read pointer always points to the location of the header flit of a packet.

5.4.5.4 Retry Example

Figure 5.8 shows the simplified block diagram of two devices, A and B, connected with an HMC link. For the purposes of retry, it does not matter which is the requester and which is the responder. The FRPs are shown being transmitted with the packets along with returning RRPs. Each device is divided into two parts, which the HMC specification refers to the link master (sender), and the link slave (receiver). In the diagram, the link master of device A is labeled MasterA, the link slave of device B is labeled SlaveB, and so forth. The steps below correspond to the numbered stars in the diagram.

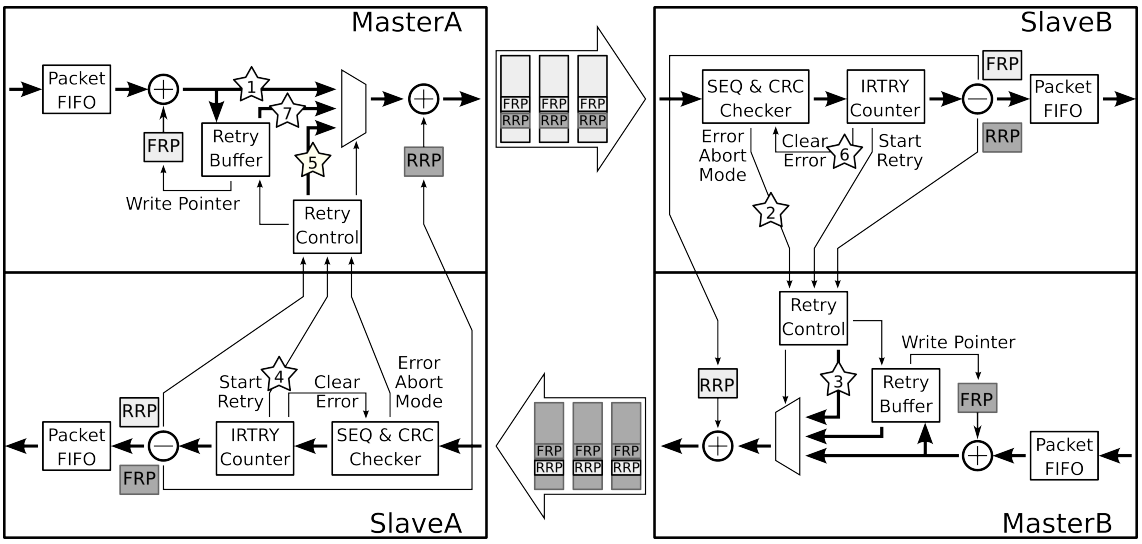

Figure 5.8: The flow of packets and information controlling retry for the HMC interface controller. Thick arrows represent packet flow. Thin arrows are signals. The numbered stars correspond to the steps listed in the text.

1. MasterA sends a packet to SlaveB.
2. SlaveB detects an error in the packet, enters error abort mode, and drops all further packets.
3. MasterB transmits a series of IRTRY packets with the StartRetry flag set and then returns to sending packets.
4. SlaveA counts consecutive IRTRY packets and signals MasterA when the threshold is reached.
5. MasterA transmits a series of IRTRY packets with the ClearErrorAbort flag set.
6. SlaveB counts consecutive IRTRY packets, exits error abort mode, and resumes normal packet processing.
7. MasterA transmits packets from the retry buffer starting at the read pointer, leaving the sequence numbers and return token count unchanged, then returns to sending packets.

There are a couple of important things to remember about IRTRY packets. The number of IRTRY packets to send and the threshold for received IRTRY packets are configurable through the register file. The HMC specification recommends setting the number of IRTRY packets to 16 so that data in a malformed packet can not be mistaken for IRTRY packets. IRTRY packets must be contiguous on the link.

IRTRY packets do not contain sequence numbers, but they do contain valid RRPs. This is important when a link error occurs in both directions so retransmission can start with the correct packet.

5.5 A Verification Environment for the HMC Interface

Debugging an HMC link controller in simulation is much simpler than debugging it in an FPGA, for several reasons: the feedback loop is shorter, test parameters are easier to control, and there is greater visibility into the state of the design when the failure occurs.

The time necessary to compile the design is one of the reasons that the feedback loop is shorter for simulation. It takes less than a minute to compile the test bench for simulation. Synthesizing, placing, and routing the design for an FPGA can take hours. An exception to this rule is the case where the error condition takes hours to reach in simulation since simulated designs run orders of magnitude slower than hardware. In a simulated environment, the tester has complete control over the inputs of the Device Under Test (DUT). This is useful for replicating input streams that produce undesired behavior. In hardware, the device receives its inputs from other components, and its functionality may be influenced by external factors such as voltage and temperature.

The internal state of the design can be explored once a failure has been isolated in simulation. The state of every register can be observed before, during, and after the failure. This allows the designer to understand if the failure is due to the design or implementation and to develop a solution. Once the design is mapped into an FPGA, the amount of internal state which is available is drastically reduced. Although there are debugging helps such as Altera's SignalTap from Xilinx's Chipscope, they are limited in the number of signals which can be captured and can affect timing closure.

In order to test the HMC controller, a Universal Verification Methodology (UVM) environment was implemented in System Verilog. The HMC BFM, available under NDA from Micron, was used in order to speed the development of the UVM components.

5.5.1 Bus Functional Model

Micron supplies a BFM for HMC controller development, subject to a NDA. The BFM implements either a requester or a responder, which are parameterized in terms

of frequency, width, and register settings. As a responder, the BFM emulates an HMC with its control and status registers.

The first step in testing the HMC controller was to connect the controller directly to the BFM, as shown in Figure 5.9. This enables early testing of the scramblers and initialization sequence with the BFM providing a useful reference point. This can then be used to provide input for testing the implementation of the UVM test code.

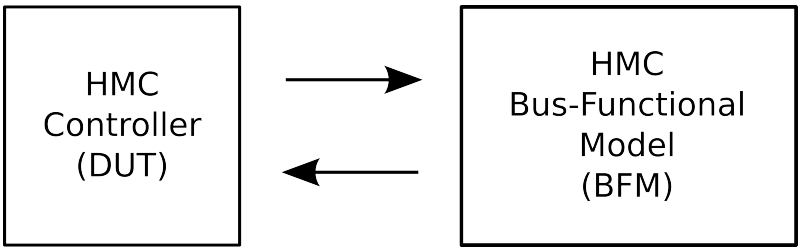

Figure 5.9: The HMC interface controller Device Under Test (DUT) can be connected directly to the Bus Functional Model (BFM).

5.5.2 The Advantages of UVM

It would also be possible to develop a HMC controller using only the Micron BFM. This could affect the development of the controller, its testing, and its distribution.

Developing an HMC controller using the BFM constrains the order in which functionality is implemented. In order to test the reception of read or write responses, for example, the transmission of requests must be implemented and functioning. Using a UVC environment separates these portions of the development, allowing the controller to be implemented in the order which is most convenient for the development team.

Using a BFM to test the HMC controller changes the way that tests are written. The purpose of a BFM is to mimic the behavior of an HMC as closely as possible. The purpose of a UVM environment is to provide coverage for all legal states of the HMC interface. This means that it is built with corner cases in mind. For a BFM, attributes such as the temporal placement of TRET flits is unimportant. The important thing is that the correct number of tokens are returned. In a UVM environment, it is also important that flits be returned at varying granularities, with different timing between TRETs, and with different delays from when the packets are received. Covering these cases makes the testing procedure more robust and improves the usability of the controller with future HMC devices which implement the HMC protocol.

One of the objectives of developing the HMC controller is to lower the barrier for others to experiment with the HMC. Developing a UVM environment makes it possible to distribute the test environment with the controller. The distribution of the test environment decreases the expected amount of support which will be required by users of the controller because it would enable them to debug the problems which they may encounter. Basing the test environment on code which is only available with an NDA would be a significant barrier to entry.

5.5.3 Architecture Overview

A simplified block diagram of the UVM environment is shown in Figure 5.10 connected to the HMC interface between the controller and the BFM. The UVM environment is self contained, so that it can be instantiated for each link in the design under test. It interacts with the link exclusively through the interface, reducing the number of side effects and making the design easier to debug. The following sections describe each part of the test environment, how they fit together, and how they communicate with one another.

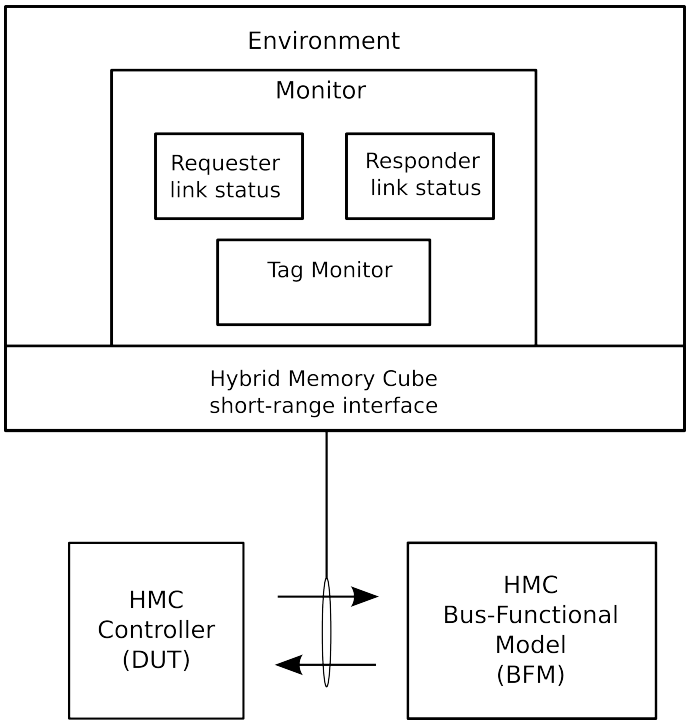

Figure 5.10: The UVM environment connects directly to the HMC short-range interface.

5.5.4 Short-Range Interface

There are two interfaces described in the HMC specification, the short-range interface and the ultra short-range interfaces. The main difference is that the short-range interface is synchronous, while the ultra-short-range interface is source-synchronous. Although this is important in terms of routing constraints and power consumption, for the test environment it just specifies whether the differential pairs should be sampled using a clock generated from the reference clock or a differential clock transmitted along with the data. The short-range interface is more flexible for prototyping, and was thus chosen for the first implementation.

The UVM interface abstracts the pins of an interface to allow other components to observe and or drive the signals of the interface. The short-range interface contains the differential pairs for transmitting and receiving data, power reduction pins, reset, and the fatal error signal. The reference clock and the pins which specify its frequency were also included for convenience.

5.5.5 Environment

The UVM Environment contains an HMC monitor, a requester agent, and a responder agent, as shown in Figure 5.11. These components share the HMC interface, and are meant to be instantiated as a group. Some parts of the requester and responder agents are drawn with dashed lines because they are only present when the agent is active, as explained in Section 5.5.7.

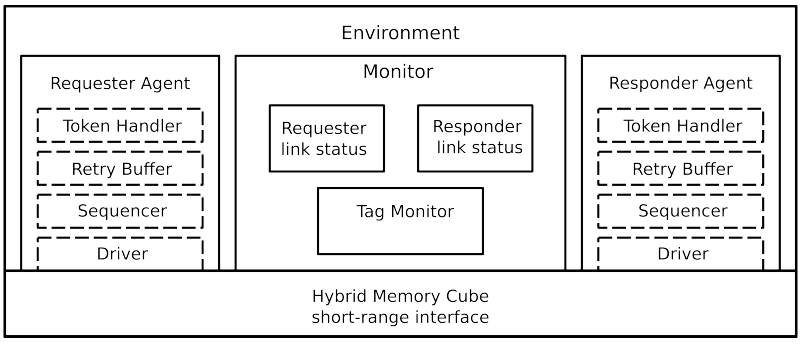

Figure 5.11: The UVM environment for the HMC interface contains a monitor, and two agents.

5.5.6 Monitor

The UVM monitor is designed to observe the interface, keeping track of its state and the packets which are exchanged. It contains a tag checker and two link-status objects. The tag checker maintains a list of the HMC tags that have been seen in request packets. When a response arrives, its tag is checked against this list. The checker aborts the simulation if tags are reused before the response arrives or a response arrives for an unused tag, and triggers an error condition when the simulation ends with tags still in the list. The link-status object is a central location to store the state of the different threads in the monitor. It also contains FIFOs to enable communication between the threads, query functions for state variables, and flow control checking.

Along with the link-status and tag-checking objects, the monitor contains multiple threads for monitoring the interface. There is a thread to monitor the power reduction pins, one to collect packets from flits, one to collect flits from the data on each lane, and one for each lane to descramble and align the data.

5.5.6.1 Descrambling

As described in Section 5.4.3.2, each lane of the HMC data is scrambled. In order to recover the data, the receiver must be XOR the scrambled data with the same value. The scrambler for each lane runs independently and, therefore, each lane needs to be descrambled independently. This function is performed by the descramble tasks in the monitor.

The descramble tasks wait for reset to be deasserted, then wait for a bit transition on their assigned lanes. After the first transition, they wait a half bit time so that the sampling point is in the center of the bit time. From this point on, they sample every bit time.

At some point, the driver on the interface will start sending scrambled zeros. The 15-bit LFSR state is then recovered by recognizing that the current state of the LFSR can be recovered from the transmitted data. Two 15-bit registers are needed, LFSR and calculatedLFSR, along with a one-bit register, lastBit, to keep track of the last value of currentBit on this lane. All of these variables can be initialized to zero. The LFSR is then recovered using the steps below.

1. Set LFSR = calculatedLFSR.
2. Set calculatedLFSR[14] = 1, to avoid locking on all zeros.

3. For the next 14 bit times (b):
    a) Set calculatedLFSR[b] equal to currentBit XOR lastBit.
    b) Set lastBit equal to currentBit.
    c) Wait one bit time.
    d) Assuming that the value in LFSR is correct, calculate the next value.
4. Compare the value of LFSR with the value of calculatedLFSR.
5. If they match, the LFSR is locked: update the link status.
6. If they do not match, go to step 1.

The important thing to understand is that the received data is the LSB of the LFSR, and the MSB is calculated using an XOR from the two LSBs. Since currentBit and lastBit are the last two LSBs, the MSB of the LFSR is just currentBit XOR lastBit. This works whether the lane is inverted or not, as can be observed by rewriting the XOR ($\oplus$) in terms of AND ($\wedge$), OR ($\vee$), and NOT ($\neg$) operations (5.9).

$$currentBit \oplus lastBit = \neg currentBit \wedge lastBit \vee currentBit \wedge \neg lastBit \quad (5.9)$$

Once the descrambler LFSR is initialized, the descrambled output is checked for lane inversion. This information is saved in the lane status by calling the corresponding functions. The task continues stepping the LFSR, descrambling the data, and waiting for non-zero bits to indicate the arrival of TS1 values, which enable flit alignment.

5.5.6.2 Flit Alignment with TS1

The HMC protocol uses a set of training (TS1) values to allow the lanes to be aligned. This is necessary since the specification allows up to 16 bit times of skew between any two lanes to ease routing constraints. The format of the TS1 values is shown in Table 5.5. Since the high-order bits are always 0xF0, this is used to align the lanes. An 8-bit register is initialized with 0xFF, and the descrambled TS1 bits are shifted in from the left. When the register contains 0xF0, the first TS1 value has been successfully received, and the lane status is updated to aligned. From this point on, partial flits are pushed into the link status FIFOs for further processing. A partial flit is the portion of a 128-bit flit transmitted by a single lane, or $128/numLanes$

Lane	position	**TS1[15:8]**	**TS1[7:4]**	**TS1[3:0]**
0	bottom	0xF0	0x3	Rolling count 0x0-0xF
1	middle		0x5	
2	middle		0x5	
⋮	⋮		⋮	
13	middle		0x5	
14	middle		0x5	
15	top		0xC	

Table 5.5: TS1 values identify lanes as top, bottom, or middle lanes with bits 7 through 4, and indicate lane alignment with bits 3 down to 0. In the case of a half-width link, lane 7 is the top lane.

bits.

5.5.6.3 Assembling Flits

Once partial flits are being collected in the link status' FIFOs, the collect-flits task assembles them into flits. After being reset, the task polls the link status to know when all lane FIFOs have a partial flit available. The first partial flits contain TS1 values, and at this point lane reversal can be detected. If 0x3 or 0xC is detected in bits seven through four of the partial flit from the top lane, the check is successful, and the lane is marked as reversed or not reversed, respectively.

The task then assembles flits and checks for NULL flits. The first NULL flit signals the beginning of valid packet transfers, and from this point on flits are sent to the flit FIFO.

5.5.6.4 Assembling Packets

The collect-packets task takes flits from the FIFO and assembles them into packets. It is written as a loop with a series of checks. At each check, the loop may be aborted. The sequence is described below.

1. Get the current flit from the FIFO.
2. If the command field is 0, go to step 1.
3. If Error Abort Mode is set and the packet is not an IRTRY, go to step 1.
4. If the length fields do not match or the length is invalid, set Error Abort Mode and go to step 1.

5. Get the next $length - 1$ flits from the FIFO.
6. Convert the flits to a packet, using the UVM unpack function.
7. If the packet contains an incorrect CRC, set Error Abort Mode and go to step 1.
8. If the packet is an IRTRY packet:
   a) Check if the last flit was also an IRTRY packet.
   b) Update the corresponding count for contiguous IRTRY packets.
   c) If the threshold is reached, send the IRTRY packet to the link status.
   d) Go to step 1.
9. If the sequence number is incorrect, set Error Abort Mode and go to step 1.
10. The packet has been checked for errors; send it to the link status.

This process filters the packets which are received and only forwards valid packets which affect the state of the link.

5.5.6.5 Link Status

The link-status object keeps track of the status of each lane and whether the link is in Error Abort Mode. These variables can be accessed through getter and setter functions. The link status also provides FIFOs for intertask communication, monitors the flow control of the link, and forwards packets and flow control information to other components.

As previously mentioned, a link-status object contains a partial flit queue for each lane and a flit queue for communication between tasks in the monitor. The maximum depth of the partial flit queues is governed by the amount of skew between lanes. Since this is limited to 16 bit times in the specification, the depth should never be more than two. The depth of the flit queue, on the other hand, should never exceed one since flits are removed at the same rate as they are inserted, as soon as they are available.

Flow control is monitored by the link-status object with a token count and a maximum token count. The maximum token count is set when the first packet which is not a TRET is received. The number of tokens is decremented when a packet is received from the monitor, and incremented when a packet is received with a non-zero token return count. The link status aborts the simulation if the number of tokens is greater than the maximum number of tokens or if packets are sent without

enough available tokens.

The link status maintains a packet FIFO to track which packets have been sent, but have not yet been acknowledged with an RRP. When packets with an FRP are received from the monitor, they are added to the FIFO. When a packet with an RRP is received, the RRP is sent to the remote link status. The remote link status then removes packets from the packet FIFO until the matching FRP is found. It is an error if there is no matching packet in the FIFO.

Information is sent from the link status in two different stages of packet processing: when packets are added to the FIFO and when they are removed from the FIFO by the receipt of an RRP. When they are received, the FRP and RRP are sent. When they have been acknowledged with an RRP, the tokens are ready for use, and the entire packet is sent. In order to simplify debugging, the entire packet is always sent, even though only certain fields are used. The following sections detail where and when the packets are sent and how the information is used.

5.5.7 Requester and Responder Agents

Once the monitor is implemented, the requester and responder agents can be implemented. When these agents are active, they replace the HMC controller or the HMC for testing. They allow finer control over packet generation, especially error condition generation, than would be possible using RTL implementations.

The environment connects the elements of the responder agent to the link status as shown in Figure 5.12. These connections are discussed further as each component of the responder agent is described.

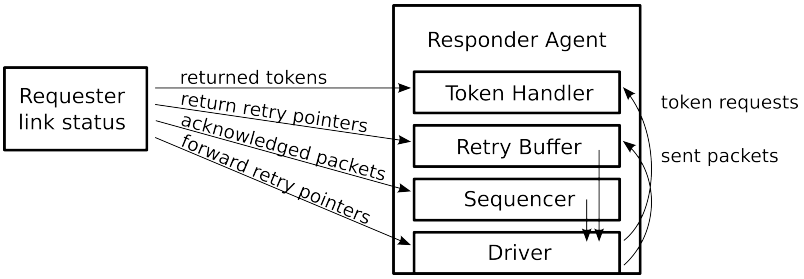

Figure 5.12: The components of the responder agent are connected to the requester link status.

The requester agent, shown with its connections to the responder link status in Figure 5.13, is very similar to the responder agent. The largest difference is the source of packets for the sequencer.

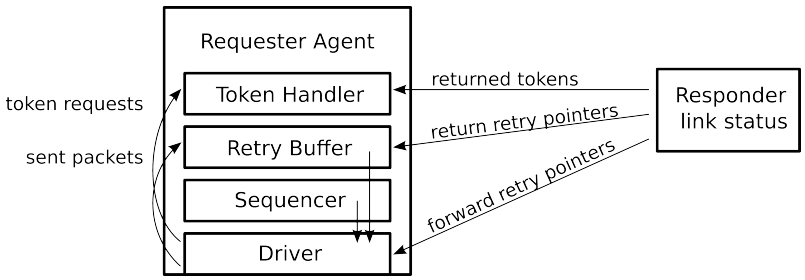

Figure 5.13: The components of the requester agent are connected to the responder link status.

5.5.7.1 Token Handler

The token handler is implemented to encapsulate flow control. It provides an interface for returned tokens and an interface to request tokens. Requests fail when there are insufficient tokens to fulfill a request.

5.5.7.2 Retry Buffer

The retry buffer is an abstract model of the buffer described in Section 5.4.5.1. It exposes three interfaces: one to add packets to the buffer, one to process RRPs and remove packets from the buffer, and one to request retry packets.

5.5.7.3 Sequencer

The sequencer provides an environment where the packet generating sequence can run. It provides a connection for receiving packets and a mailbox to communicate with the sequencer. It simply puts received packets into the mailbox and fails if the mailbox is full.

5.5.7.4 Sequence

A sequence generates packets for transmission over the link. It has an executable body and uses the *uvm_send*() macro to forward the packets it has generated. In this section the responder sequence is described since it is more complex than the requester sequence.

The responder sequence consists of two asynchronous loops, which communicate via an event and a list of packets sorted by timestamp. The first loop waits for a request packet to be written to the mailbox, translates it into a response, inserts it

into a sorted list, and uses an event to signal that a response has been added to the list.

The second loop waits for the event and checks the timestamp of the first packet in the list. If the current time is less than the timestamp, it waits until the timestamp is reached. At this point, it sends the packet with the smallest timestamp. Note that this packet may be different than the original packet since packets can be added while the loop waits.

The two loops are needed because the second loop sleeps until the smallest timestamp is reached, and because the *uvm_send*() macro contains a call which does not return until the packet is successfully sent. Having the loops running independently removes the possibility of losing packets while the thread is sleeping or blocked.

5.5.7.5 Driver

The function of a driver is to translate a packet from the sequencer to logic levels on an interface.

The largest difference between the requester and receiver drivers is the during initialization, which is illustrated in Figure 5.14. The responder passes through two more initialization states than the requester: a high-impedance state and a PRBS state. The conditions for and the timing of state transitions are also different.

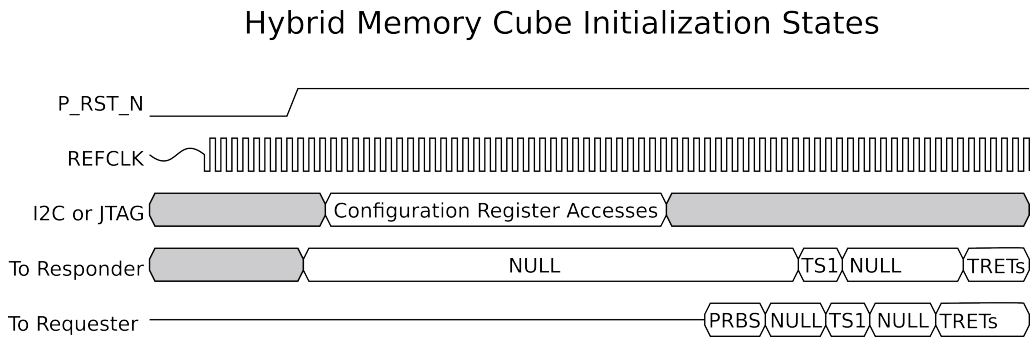

Figure 5.14: A simplified timing diagram for the initialization of the HMC link showing the states through which the requester and the responder pass.

Once the HMC interface is initialized, the only difference between the requester and responder drivers is whether the receive or the transmit pins of the interface are driven.

Since the requester and responder drivers are very similar, they inherit from a common base driver class. The base driver implements scrambling, flow control, packet sending, and retry. The requester and responder implement reset, the link

state machine, unique initialization states, and the function that drives flits onto the interface.

Now that each component of the driver has been introduced, the connections shown in Figure 5.12, included again as 5.15 for ease of reference, are more easily understood.

The token handler is connected to the link status and the driver. It receives returned tokens from the link status and responds to the driver's requests.

The retry buffer has two connections to the driver and one to the link status. It receives packets which will be driven onto the interface for possible retransmission and provides these packets to the driver during retry, in their original order. It uses the RRP values received from the link status to remove packets from the retry buffer.

The sequencer receives packets which have been acknowledged and removed from the retry buffer from the link status. This ensures that a response will never be generated for a packet which has not been successfully received. It sends generated packets to the driver for transmission.

The driver is connected to the link status and to the other three components of the agent. It receives IRTRY packets and FRPs from the link status. The IRTRY packets control retry, and the FRPs are embedded as RRPs in packets to be returned. Its connections to the other components were described with the component descriptions.

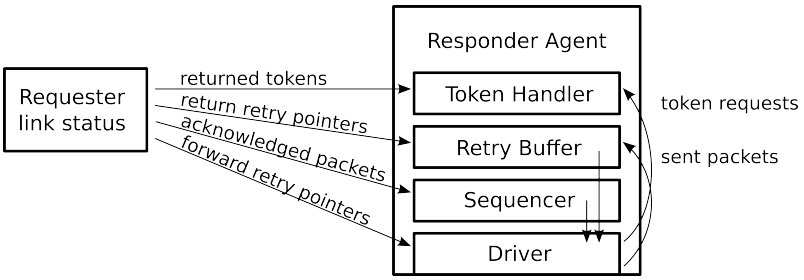

Figure 5.15: The requester link status sends tokens to the token handler, Return Retry Pointers (RRPs) to the retry buffer, acknowledged packets to the responder sequencer, and Forward Retry Pointers (FRPs) to the driver.

Another important function of the driver is to model the effects of the physical connection. This includes bit errors, routing delays, lane reversal, and lane polarity reversal. This is done for convenience in creating tests. All of these effects could also be tested using a model of the physical medium, but the correlation with how the errors affected the packets would be lost.

5.5.8 UVM Conclusion

A UVM test framework was built to enable directed testing of an HMC design. Using the BFM to generate valid traffic allowed the monitor to be built first. Once the monitor was functional, the requester and responder drivers were built using information from the monitor's link-status objects. At this point, the BFM is not necessary for further development.

5.6 Prototyping Complexity

Although the HMC is a simple flit-based protocol, its high frequency increases the complexity of implementing a controller in an FPGA. It has a minimum bit rate of 10Gb/s, which translates to a minimum bandwidth of 10GB/s for a half-width (8-lane) design. Since the difficulty of an FPGA implementation increases with the target frequency, much of the complexity of the design owes to the high serialization and deserialization factor necessary to keep the design under 250MHz.

5.6.1 High Serialization and Deserialization Factor

Table 5.6 shows the design space for the FPGA datapath widths and frequencies. FPGAs commonly support serialization factors which are multiples of 8 and multiples of 10, in order to provide support for 8B10B encoded protocols. The first implementation is an 8-lane implementation, but the datapath parameter values for a 16-lane implementation are also shown.

The first factor for which the datapath frequency is under 250 MHz is 64, which leads to a 512-bit wide datapath at 156.25 MHz. If it had not been for the increased complexity of handling half-flits, 40 would have been considered since the criteria under 250 MHz is somewhat arbitrary.

Using even larger factors was not considered because of the drawbacks of wider datapaths. Along with higher logic utilization and power consumption, wider datapaths increase the logic complexity. This is easy to see for the scrambler and descrambler, where the number of bits that need to be calculated is directly correlated with the number of XOR gates. In that case, however, it is expressed with a generate loop, grows linearly with width, and requires little extra effort to implement or debug.

Some of the logic complexity grows superlinearly with the width of the datapath. Two interrelated problems are: finding packets in the data stream and checking their

Lanes	Factor	Width	Flits	Frequency at 10 Gb/s	Frequency at 15 Gb/s
8	16	128	1	625	937.5
	20	160	1.25	500	750
	32	256	2	312.5	468.75
	40	320	2.5	250	375
	64	512	4	156.25	234.375
	80	640	5	125	187.5
16	8	128	1	1300	1875
	16	256	2	625	937.5
	20	320	2.5	500	750
	32	512	4	312.5	468.75
	40	640	5	250	375
	64	1024	8	156.25	234.375
	80	1280	10	125	187.5

Table 5.6: The possible datapath widths and frequencies based on the serialization factor chosen in the FPGA.

CRC values. Figure 5.16 shows the possible combinations of flits for a datapath width of one, two, and three flits. As previously noted in Figure 5.5, there are only four valid flit types. It should be noted that the number of possibilities doubles with each additional flit. This is due to the fact that flits end with either a tail or data. If a flit ends with a tail, the next flit must contain a header. If a flit ends with data, the next flit must start with data. For each of these cases, there are two possible flits.

This means that with a single-flit architecture, there will be at most one packet, and therefore one CRC calculation and location, at a time. For architectures with wider datapaths, the number of simultaneous CRC calculations, the possible locations of headers and tails, and the number of possible cases grows. This significantly increases the amount of effort needed to design and debug the link controller.

5.6.2 Bit Transition Count

Another feature that scales poorly is the bit transition count logic. This is needed to enforce the per-lane run-length limitation of 85 bits. As explained in Section 5.4.3.1, if there are insufficient transitions in the data stream, a bit must be flipped so that the clock recovery circuit will not lose its lock. This requires keeping a count of consecutive bits without a transition for each lane. The higher the serialization factor, the more bits must be counted, and the more locations might need to be

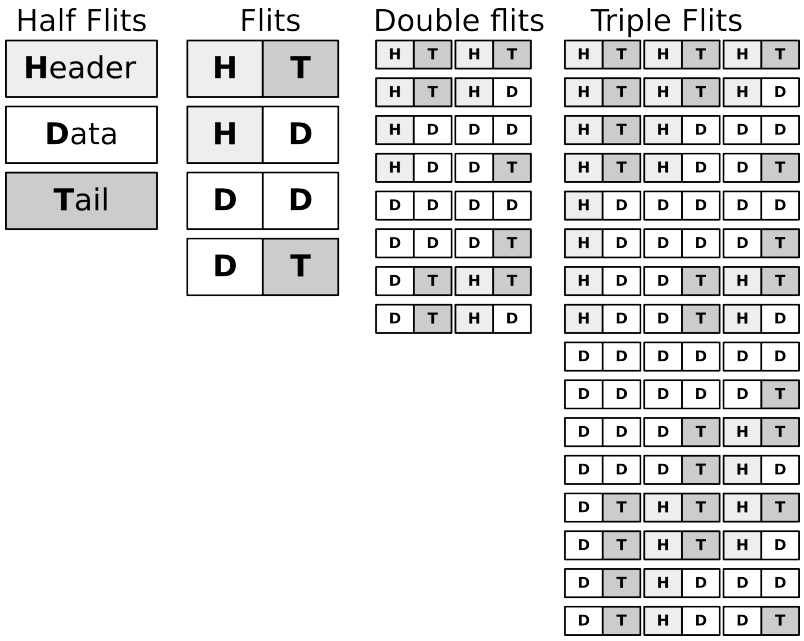

Figure 5.16: The possible arrangements of flits for datapath widths of one, two, and three flits.

flipped.

In order to simplify this logic, the 8-lane implementation with a serialization factor of 64 always flips the first bit of the outgoing data. Since this logic is replicated per lane, every 16 bits is a new flit, and more flits than necessary will follow the inserted error. For example, if 21 bits from the previous cycle and 64 bits from the current cycle contain no transition, a transition will be inserted in the first flit from the current cycle. The two following flits could have been successfully transmitted, but will also be invalidated. In practice, the run-length limit should be encountered so infrequently that this is unimportant.

5.7 Scalable Packet Processing Architecture

The initial version of the HMC controller core is designed specifically for 8 lanes and a deserialization factor of 64. Since the number of flits does not change, the design could also work for a 16-lane design with a deserialization factor of 32 if the frequency target of 312.5 MHz could be reached. Porting the design to an ASIC or another FPGA architecture with a different datapath width would require substantial redesign. This section describes a packet processing architecture that can be more easily adapted to different datapath widths and therefore reused for

various technologies. This should reduce the testing and verification effort since the majority of the code is shared.

5.7.1 Overview

The basic idea of the architecture is to trade simplicity and scalability for latency, as measured in clock cycles. If a block can be designed which efficiently processes single packets, multiple packets can be processed by connecting these blocks in parallel. The goal is to avoid the superlinear increase in complexity that comes with multiple-flit datapaths. Reducing the complexity should result in a higher clock frequency, and reduce the penalty of the introduced pipeline stages. It is important to note that the width of the pipeline is unaffected, and that no further clock domains are introduced.

The architecture is divided into three phases: stream splitting, packet checking, and stream reassembly, as shown in the architectural block diagram in Figure 5.17. This figure illustrates the flow of packets through a three-flit datapath. The receiver (Rx) and transmitter (Tx) datapaths make use of the concept in a similar way, so the Rx datapath is described first, then a shorter description of the differences in the Tx implementation is given.

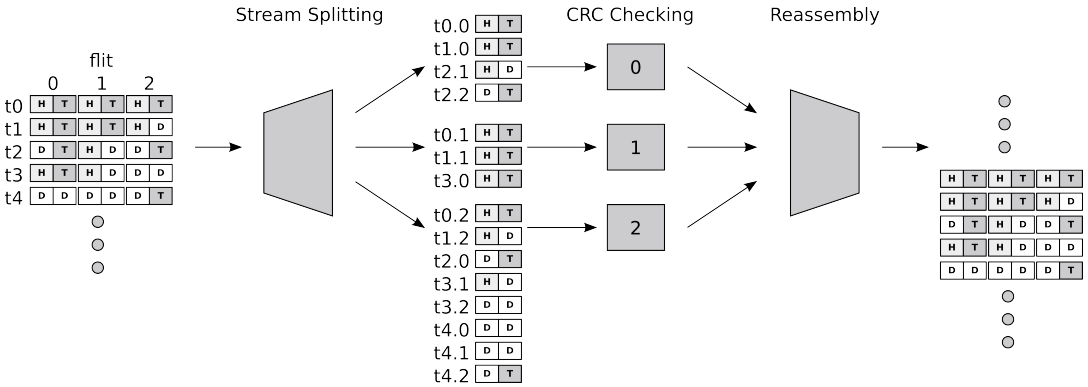

Figure 5.17: A block diagram of the scalable packet processing architecture, showing the three logical phases.

5.7.2 Receiver Stream Splitting

The first stage of the scalable architecture comes after the descramblers. At this point, the data is aligned in flits, but the packet locations are unknown. The stream splitter searches the input stream for non-null flits, reads and checks length fields, and assigns the packet to a packet checker for CRC checking and further processing. A form of round-robin scheduling is preferred since the order of the packets is important

in the HMC protocol. This stage of the pipeline is still dependent on the width of the datapath and will, therefore, require modifications when changing widths. The complexity of stream splitting grows linearly with the number of flits since the number of possible packets is equal to the number of flits.

Figure 5.17 also illustrates the problem with using strict round-robin scheduling. In this example, packet checker number 2 is assigned eight flits in the same time that checkers 0 and 1 are assigned 4 and 3, respectively. This leads to wasted resources at some units and overflows at others. The splitter routine must therefore take the available buffer space into account when assigning packets.

5.7.3 Packet Checkers with CRC

The packet checker contains the logic that scales superlinearly with datapath width and is therefore implemented for a single flit width, for maximum flexibility. For example, referring to Table 5.6, a single-flit packet checker could be implemented for use in an 8-lane, 10 Gb/s design in an ASIC running at 625 MHz. A 16-lane design could be served by instantiating two single-flit checkers and sending each of them alternating packets. The CRC check is implemented at this stage, but other correctness checks such as packet sequence numbers are delayed until the CRC is verified.

Each CRC checker needs to be able to buffer a complete packet at its input and another at its output for this to work since packets can be various lengths and shorter packets take less time to process. The important thing is to maintain high throughput to avoid overflow. At this point, it is clear that latency (in terms of cycles) has to be sacrificed to allow a single-flit CRC to serve wider datapaths.

The most significant difference for prototyping is that the number of cases for testing is dramatically reduced. The architecture is promising because there is minimal overhead for an ASIC implementation, but as the width of the datapath increases, the overhead increases.

5.7.4 Receiver Stream Reassembly

After the packets have had their CRC values checked, they must be reassembled into a packet stream to maintain their original order. This means that packets must be removed from the CRC checkers in the same fashion as they were inserted. Some extra information about where packets start and stop is kept to speed further processing.

At this point, the rest of the packet checking is implemented: sequence numbers are checked, flow information is collected, and flow packets are discarded. When an error is detected, all packets which follow are discarded.

The request and response packets continue on to the input buffer where they are stored until they can be processed or sent on.

5.7.5 Applying the Architecture to the Transmitter

The components and their connections are very similar in the Tx direction. Packets still flow in and out in a stream. The stream is split and reassembled in a similar fashion. The main difference is that the pipeline stage where the CRC values are compared in the Rx design becomes the stage where the CRC value is inserted for Tx. After the stream is reassembled, the flit stream is separated into lanes and sent to the scramblers and the serializers.

5.7.6 Scalable Architecture Summary

Isolating and encapsulating the logic portions that scale nonlinearly increases the flexibility of the design. It opens up the possiblity of designing efficient hardware for ASIC implementations and reusing multiple instances of that logic in parallel for prototyping at lower frequencies. This reduces the number of corner cases, which must be covered in testing, and should reduce the amount of time which is needed to port the HMC controller to a new architecture. Although the architecture is sketched out, it is not yet implemented. The implementation and performance comparison with the current architecture is left for future work.

5.8 Future Narrow Memory Interfaces

The HMC protocol is promising because of its simplicity, relative ease of implementation, and level of abstraction. It is easy to imagine that if the HMC protocol does not achieve wide-spread adoption, another narrow interface will. The HMC is not the first attempt to build a narrow memory interface, although the need has been recognized for some time. Two areas of focus for a new interface could be addressing and networking.

5.8.1 Address Bits

One of the most obvious drawbacks to the HMC protocol is the limited number of address bits per cube. The 34 address bits provide byte addressing capabilities for 16GB of storage per cube. The first HMC devices have a capacity of 2GB, with 4GB devices already announced. Besides restricting the longevity of the HMC protocol for use with DRAM, this has consequences for its use as a generalized storage network. Two of the reasons that the HMC protocol has a limited number of address bits are reserved bits and byte addressability.

There are three bits in request packet headers that are reserved for future expansion of the cube ID or the address. Expanding the cube ID field increases the number of HMCs that can be interconnected, while expanding the address increases the possible capacity of each node. The applications which make use of the HMC will likely dictate where the extra bits will be used.

The other reason for limited address bits is byte addressability. The minimum granularity is 16 bytes for read and write requests, 8 bytes for add immediate and bit writes, and 4 bytes for mode and status register accesses. This means that for the majority of accesses, the lower four address bits are zero. These bits could be easily reclaimed in future versions of the HMC protocol. The address is only used at the destination, and multiple decoding schemes are already supported based on the choice of the maximum block size.

5.8.2 Networking Possibilities

The HMC protocol provides many of the same features which are needed in order to build a high-performance network. Flow control, packet routing, and error checking features are all present. Communications protocols could be implemented using read and write commands, or additional commands using reserved command combinations, but the HMC protocol only supports 8 nodes. It is then interesting to ask what differentiates the HMC protocol from other high-bandwidth network protocols and whether they will converge in one direction or the other.

Some of the possible networking features include:

- virtual channels for deadlock avoidance
- adaptive routing
- network discovery

5.9 Conclusion

The memory interface is the last wide interface in the personal computer which has not been replaced by a narrow one. The HMC is a possible contender for new memory standard. It is unclear whether the HMC will be widely accepted, but it is likely that future upgradeable memory systems will have a narrow interface.

6 Conclusion

This work motivated the adoption of packetized memory interfaces to enable new memory hierarchies, ease prototyping, and provide flexibility for future technology developments. This was supported by multiple projects employing packetized memory interfaces and novel architectures.

There are many competing solid state memory technologies. This work presented an overview starting with DRAM and SRAM. It also discussed non-volatile memory technologies, including EEPROM, NOR and NAND flash, and several emerging technologies, including PCM, RRAM, and FRAM. Multiple technologies could scale to the point at which they could be used to implement non-volatile main memories. Instead of presenting a prediction for which technology will win, this work posits that the important thing to recognize is that there are multiple technologies which may be used in future memory hierarchies. This uncertainty impedes the adoption of memory technologies since their interfaces are incompatible. One of the contributions of this work is the presentation of the argument that the choice of a storage technology can be a secondary consideration, if a packetized memory interface is employed. An added benefit of this flexibility is the ability to create heterogeneous memory hierarchies in order to optimize performance and power consumption.

Another reason for packetized memory interfaces is an increased flexibility for prototyping. Thus, one contribution of this work is the implementation of DiskRAM, a prototype memory controller. DiskRAM performs paging in hardware and presents a large flat address space to the processor. The implementation of DiskRAM was only possible because the HyperTransport bus in a multiprocessor system can also be used as a memory interface. This packetized interface provided the timing flexibility necessary to allow the prototype to process and respond to requests. If there had been a packetized memory interface available for prototyping DiskRAM, it would have made it considerably easier to prototype. A packetized memory interface would have been free of coherency traffic and would have required much less adaptation in the configuration and initialization of the system.

After advocating the adoption of packetized memory interfaces for single nodes,

this work presented the NAM, a device for adding globally addressable memory to distributed machines. This memory is designed to be used to increase the performance of I/O operations and checkpointing. The packetized network interface offers the flexibility to model multiple types of memory with various timing differences and to distribute the memory throughout the cluster. The packetized interface also provides the ability to implement global operations in the network.

The final contribution of this work is the discussion of the importance of the abstraction provided by packetized memory interfaces. The HMC interface, which was developed for stacked DRAM, was evaluated in this context. Much of the knowledge presented was gained while implementing a Universal Verification Methodology (UVM) environment for testing and debugging an HMC controller. The abstraction of the interface motivated the comparison of the HMC interface to a network interface, and pointed out that these two interfaces may converge in the future.

6.1 Future Work

There are many ways in which the research presented in this work can be continued. One of the most interesting is reevaluating the idea of demand paging in hardware with solid-state storage instead of hard disk drives. Non-homogeneous stacked storage would make it possible to implement a non-volatile storage with a volatile cache that was transparent to the processor. Such a memory hierarchy would save power by optimizing data placement and increase performance by recognizing dynamic usage patterns.

There is more work to be done with the NAM. The most obvious place to start is the use of benchmarks to characterize its performance and the comparison of these numbers to parallel file systems. Using this data, the number of NAMs that should be added to distributed systems and their effects on the design of the parallel file system is worth investigating. Another area of work is to use a set of parallel workloads to extract the global operations worth accelerating using the NAM. The effect of global operations in the NAM on network traffic patterns is also interesting.

Future work on abstract memory interfaces should build on the experience gained from using the Hybrid Memory Cube interface to improve upon it. One interesting avenue to pursue is the design of a unified network and memory interface. This would allow the creation of easily customizable distributed computers. The next step is to make such a customizable computer dynamically reconfigurable. This reconfigurability would be useful for balancing power consumption and performance

in a workload-specific manner.

The NAM architecture developed in this work provides the experimental foundation for exploring such innovative uses for packetized memory interfaces.

Bibliography

[1] G. E. Moore, "Lithography and the Future of Moore's Law", in *Proceedings of the SPIE*, R. D. Allen, Ed., vol. 2438, Jun. 1995, pp. 2–17.

[2] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. Leblanc, "Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions", in *Proceedings of the IEEE*, vol. 87, Apr. 1999, pp. 668–678.

[3] S. Borkar, "Getting Gigascale Chips", *Queue*, vol. 1, no. 7, p. 26, Oct. 2003.

[4] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. 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*, 2010, pp. 175–186.

[5] JEDEC Solid State Technology Association, *DDR3 SDRAM Standard*, 2012.

[6] Integrated Circuit Engineering Corporation, "SRAM Technology", in *Memory 1997*, 1997, ch. 8.

[7] ——, "DRAM Technology", in *Memory 1997*, X, 1997, ch. 7.

[8] ——, "Flash memory technology", in *Memory 1997*, 1997, ch. 10.

[9] R. Sbiaa, H. Meng, and S. N. Piramanayagam, "Materials with Perpendicular Magnetic Anisotropy for Magnetic Random Access Memory", *Physica Status Solidi (RRL) - Rapid Research Letters*, vol. 5, no. 12, pp. 413–419, Dec. 2011.

[10] N. N. Mojumder, X. Fong, C. Augustine, S. K. Gupta, S. H. Choday, and K. Roy, "Dual Pillar Spin-Transfer Torque MRAMs for Low Power Applications", *ACM Journal on Emerging Technologies in Computing Systems*, vol. 9, no. 2, pp. 1–17, May 2013.

[11] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative", in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, Apr. 2013, pp. 256–267.

[12] *International Technology Roadmap for Semiconductors*, 2011.

[13] C. R. McWilliams, J. Celinska, C. a. Paz de Araujo, and K.-H. Xue, "Device Characterization of Correlated Electron Random Access Memories", *Journal of Applied Physics*, vol. 109, no. 9, p. 091 608, 2011.

[14] M. J. Sánchez, M. J. Rozenberg, and I. H. Inoue, "A Mechanism for Unipolar Resistance Switching in Oxide Nonvolatile Memory Devices", *Applied Physics Letters*, vol. 91, no. 25, p. 252 101, 2007.

[15] B. Liu, J. F. Frenzel, and R. B. Wells, "A Multi-Level DRAM with Fast Read and Low Power Consumption", in *Proceedings of the IEEE Workshop on Microelectronics and Electron Devices. WMED '05.*, IEEE, 2005, pp. 59–62.

[16] J. C. Koob, S. A. Ung, B. F. Cockburn, and D. G. Elliott, "Design and Characterization of a Multilevel DRAM", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 9, pp. 1583–1596, Sep. 2011.

[17] G. Birk, D. Elliott, and B. Cockburn, "A comparative simulation study of four multilevel DRAMs", in *Records of the 1999 IEEE International Workshop on Memory Technology, Design and Testing*, IEEE Comput. Soc, 1999, pp. 102–109.

[18] M. Aoki, Y. Nakagome, M. Horiguchi, S. Ikenaga, and K. Shimohigashi, "A 16-Level/Cell Dynamic Memory", *IEEE Journal of Solid-State Circuits*, vol. 22, no. 2, pp. 297–299, Apr. 1987.

[19] S. Lee, Y.-T. Lee, W.-K. Han, D.-H. Kim, M.-S. Kim, S.-h. Moon, H. C. Cho, J.-W. Lee, D.-S. Byeon, Y.-H. Lim, H.-S. Kim, S.-H. Hur, and K.-D. Suh, "A 3.3 V 4 Gb Four-Level NAND Flash Memory with 90 nm CMOS Technology", in *2004 IEEE International Solid-State Circuits Conference*, vol. 36, IEEE, 2004, pp. 52–513.

[20] D. H. Yoon, J. Chang, R. S. Schreiber, and N. P. Jouppi, "Practical Nonvolatile Multilevel-Cell Phase Change Memory", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '13*, New York, New York, USA: ACM Press, 2013, pp. 1–12.

[21] N. H. Seong, S. Yeo, and H.-H. S. Lee, "Tri-Level-Cell Phase Change Memory", in *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*, New York, New York, USA: ACM Press, 2013, p. 440.

[22] G. Roos and B. Hoefflinger, "Three-Dimensional CMOS NAND with Three Stacked Channels", *Electronics Letters*, vol. 29, no. 24, pp. 24–25, 1993.

[23] H.-T. Lue, T.-H. Hsu, Y.-H. Hsiao, S. P. Hong, M. T. Wu, F. H. Hsu, N. Z. Lien, S.-Y. Wang, J.-Y. Hsieh, L.-W. Yang, T. Yang, K.-C. Chen, K.-Y. Hsieh, and C.-Y. Lu, "A Highly Scalable 8-Layer 3D Vertical-Gate (VG) TFT NAND Flash Using Junction-Free Buried Channel BE-SONOS Device", *2010 Symposium on VLSI Technology Digest of Technical Papers*, pp. 131–132, Jun. 2010.

[24] Y.-H. Hsiao, H.-T. Lue, T.-H. Hsu, K.-Y. Hsieh, and C.-Y. Lu, "A Critical Examination of 3D Stackable NAND Flash Memory Architectures by Simulation Study of the Scaling Capability", in *Proceedings of the 2010 IEEE International Memory Workshop*, IEEE, 2010, pp. 1–4.

[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Second Edition. Elsevier, 1996.

[26] P. J. Denning, "Virtual Memory", *ACM Computing Surveys*, vol. 2, no. 3, pp. 153–189, Sep. 1970.

[27] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic Tracking of Page Miss Ratio Curve for Memory Management", in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS-XI*, vol. 32, New York, New York, USA: ACM Press, Dec. 2004, p. 177.

[28] F. G. Soltis, *Inside the AS/400*. 29th Street Press, 1996.

[29] J. Wawrzynek, D. Patterson, M. Oskin, S.-l. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research Accelerator for Multiple Processors", *IEEE Micro*, vol. 27, no. 2, pp. 46–57, Mar. 2007.

[30] D. Slogsnat, A. Giese, M. Nüssle, and U. Brüning, "An open-source HyperTransport core", *ACM Transactions on Reconfigurable Technology and Systems*, vol. 1, no. 3, pp. 1–21, Sep. 2008.

[31] H. Fröning, M. Nüssle, D. Slogsnat, H. Litz, and U. Brüning, "The HTX-board: a rapid prototyping station", in *3rd annual FPGAWorld Conference*, Stockholm, Sweden, 2006.

[32] D. Slogsnat, A. Giese, and U. Brüning, "A versatile, low latency HyperTransport core", *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field programmable Gate Arrays - FPGA '07*, p. 45, 2007.

[33] M. Watson and K. Flanagan, "System-level Prototyping with HyperTransport", in *Proceedings of the Second International Workshop on HyperTransport Research and Applications (WHTRA2011)*, 2011, pp. 1–6.

[34] S. Kapferer, "Design Space Analysis and Implementation of a Cache Coherent Device for HyperTransport", Diplom, University of Mannheim, 2007, pp. 1–94.

[35] Ubuntu. (2014). Ubuntu: The Leading OS for PC, Tablet, Phone and Cloud, [Online]. Available: `http://www.ubuntu.com/`.

[36] DSL. (2014). Damn Small Linux, [Online]. Available: `http://www.damnsmalllinux.org/`.

[37] DEEP. (2014). Dynamical Exascale Entry Platform, [Online]. Available: `http://www.deep-project.eu/`.

[38] D. A. Mallon, N. Eicker, M. E. Innocenti, G. Lapenta, T. Lippert, and E. Suarez, "On the Scalability of the Clusters-Booster Concept", in *Proceedings of the Future HPC Systems on the Challenges of Power-Constrained Performance - FutureHPC '12*, New York, New York, USA: ACM Press, 2012, pp. 1–10.

[39] DEEP-ER. (2014). Dynamical Exascale Entry Platform - Extended Reach, [Online]. Available: `http://www.deep-er.eu/`.

[40] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward Exascale Resilience", *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, Sep. 2009.

[41] A. Moody, G. Bronevetsky, K. Mohror, and B. R. D. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '10*, 2010.

[42] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing Checkpoints Using NVM as Virtual Memory", *Proceedings of the 27th International Symposium on Parallel and Distributed Processing*, pp. 29–40, May 2013.

[43] X. Li, K. Lu, X. Wang, and X. Zhou, "NV-process: A Fault-Tolerance Process Model Based on Non-Volatile Memory", in *Proceedings of the Asia-Pacific Workshop on Systems - APSYS '12*, New York, New York, USA: ACM Press, 2012, pp. 1–6.

[44] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories", in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, Dec. 2010, pp. 385–395.

[45] K. Grimsrud and H. Smith, *Serial ATA Storage Architecture and Applications: Designing High-Performance, Cost-Effective I/O Solutions*. Intel Press, 2003, p. 2.

[46] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling", in *Proceedings of the 38th Annual International Symposium on Computer Architecture - ISCA '11*, New York, New York, USA: ACM Press, 2011, p. 365.

[47] J. Meza, J. Li, and O. Mutlu, "A Case for Small Row Buffers in Non-Volatile Main Memories", in *Proceedings of the 30th International Conference on Computer Design (ICCD)*, IEEE, Sep. 2012, pp. 484–485.

[48] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated Control for Energy-Efficient and Heterogeneous Memory Systems", in *Proceedings of the19th International Symposium on High Performance Computer Architecture (HPCA)*, Ieee, Feb. 2013, pp. 424–435.

[49] E. Cooper-Balis, P. Rosenfeld, and B. Jacob, "Buffer-On-Board Memory Systems", in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, Jun. 2012, pp. 392–403.

[50] *Hybrid Memory Cube Specification 1.0*, 2013.

[51] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance", in *Proceedings of the Symposium on VLSI Technology (VLSIT)*, IEEE, Jun. 2012, pp. 87–88.

[52] S. Spiesshoefer, L. Schaper, S. Burkett, G. Vangara, Z. Rahman, and P. Arunasalam, "Z-Axis Interconnects Using Fine Pitch, Nanoscale Through-Silicon Vias: Process Development", in *Proceedings of the 54th Electronic Components and Technology Conference*, IEEE, 2004, pp. 466–471.

[53] S. Seetharam, G. Minden, and J. Evans, "A Parallel SONET Scrambler/Descrambler Architecture", in *Proceedings of the International Symposium on Circuits and Systems*, IEEE, 1993, pp. 2011–2014.

[54] I. V. Denisov, O. V. Kirichenko, V. A. Sedov, V. V. Vorobyev, A. V. Artemyev, and R. S. Drozdov, "One-Wire Network Standard for the Fiber Optic Measuring Network", in *Proceedings of SPIE 5135, Optical Information, Data Processing and Storage, and Laser Communication Technologies*, J.-P. Goedgebuer, N. N. Rozanov, S. K. Turitsyn, A. S. Akhmanov, and V. Y. Panchenko, Eds., Sep. 2003, pp. 5–13.

[55] P. Koopman, "32-Bit Cyclic Redundancy Codes for Internet Applications", in *Proceedings of the International Conference on Dependable Systems and Networks*, IEEE Comput. Soc, 2002, pp. 459–468.

[56] J. Ray and P. Koopman, "Efficient High Hamming Distance CRCs for Embedded Networks", in *Proceedings of the International Conference on Dependable Systems and Networks*, IEEE, 2006, pp. 3–12.