OPPORTUNITIES FOR NEAR DATA COMPUTING IN MAPREDUCE WORKLOADS

by

Seth Hintze Pugsley

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2015

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Seth Hintze Pugsley**

has been approved by the following supervisory committee members:

Rajeev Balasubramonian	, Chair	**12 May 2014** Date Approved
Alan L. Davis	, Member	**12 May 2014** Date Approved
Erik L. Brunvand	, Member	**3 June 2014** Date Approved
Feifei Li	, Member	**15 May 2014** Date Approved
Vijayalakshmi Srinivasan	, Member	**19 June 2014** Date Approved

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

In-memory big data applications are growing in popularity, including in-memory versions of the MapReduce framework. The move away from disk-based datasets shifts the performance bottleneck from slow disk accesses to memory bandwidth. MapReduce is a data-parallel application, and is therefore amenable to being executed on as many parallel processors as possible, with each processor requiring high amounts of memory bandwidth. We propose using Near Data Computing (NDC) as a means to develop systems that are optimized for in-memory MapReduce workloads, offering high compute parallelism and even higher memory bandwidth. This dissertation explores three different implementations and styles of NDC to improve MapReduce execution. First, we use 3D-stacked memory+logic devices to process the Map phase on compute elements in close proximity to database splits. Second, we attempt to replicate the performance characteristics of the 3D-stacked NDC using only commodity memory and inexpensive processors to improve performance of both Map and Reduce phases. Finally, we incorporate fixed-function hardware accelerators to improve sorting performance within the Map phase. This dissertation shows that it is possible to improve in-memory MapReduce performance by potentially two orders of magnitude by designing system and memory architectures that are specifically tailored to that end.

Dedicated to my wife, Melina.

CONTENTS

ABSTRACT . . . . . **iii**

LIST OF TABLES . . . . . **ix**

ACKNOWLEDGEMENTS . . . . . **x**

CHAPTERS

1. INTRODUCTION **1**

1.1 Emerging Trends 1
1.1.1 Big Data 1
1.1.2 MapReduce 2
1.1.3 In-Memory Databases 2
1.1.4 Commodity Servers 3
1.1.5 3D-Stacked Manufacturing 3
1.2 Dissertation Overview 4
1.2.1 Thesis Statement 5
1.2.2 Near Data Computing with 3D-Stacked Memory+Logic Devices 5
1.2.3 Near Data Computing Using Commodity Memory 5
1.2.4 Near Data Computing and Fixed Function Accelerators 6

2. BACKGROUND **7**

2.1 MapReduce 7
2.1.1 Map 7
2.1.1.1 Map Scan 8
2.1.1.2 Sort 8
2.1.1.3 Combine 9
2.1.1.4 Partition 9
2.1.2 Shuffle and Sort 9
2.1.3 Reduce 10
2.2 3D Stacking 10
2.2.1 Through-Silicon Vias 10
2.2.2 Hybrid Memory Cube 11
2.2.2.1 DRAM Organization 12
2.2.2.2 Logic Layer 12
2.3 Processing in Memory 13
2.3.1 DIVA 14
2.3.2 Vector IRAM 15
2.3.3 PIM-Lite 15

2.3.4 FlexRAM 15
2.3.5 Gilgamesh 15
2.3.6 Mercury 15
2.3.7 Other Related PIM Work 16
2.4 Hardware Accelerators for Big Data Applications 16
2.4.1 Thin Servers with Smart Pipes 17
2.4.2 HARP 17
2.4.3 Q100 17

3. NEAR DATA COMPUTING WITH 3D-STACKED MEMORY+LOGIC DEVICES 19

3.1 Memory System Background 20
3.1.1 Moving from DDR3 to HMC 20
3.1.2 Analyzing an HMC-Based Design 20
3.2 Near Data Computing Architecture 21
3.2.1 High Performance Baseline 21
3.2.2 NDC Hardware 23
3.2.2.1 3D NDC Package 24
3.2.2.2 3D Vertical Memory Slice 24
3.2.2.3 NDCores 25
3.2.2.4 Host CPUs and 3D NDC Packages 25
3.2.2.5 Power Optimizations 26
3.2.3 NDC Software 26
3.2.3.1 User Programmability 26
3.2.3.2 Data Layout 27
3.2.3.3 MapReduce Runtime 28
3.3 Evaluation 28
3.3.1 Evaluated Systems 28
3.3.1.1 OoO System 29
3.3.1.2 EECore System 29
3.3.1.3 NDCore System 29
3.3.2 Workloads 29
3.3.3 Methodology 30
3.4 Performance Results 31
3.4.1 Individual Mapper Performance 31
3.4.2 Map Phase Performance 32
3.4.3 Bandwidth 34
3.4.4 MapReduce Performance 35
3.4.5 Energy Consumption 36
3.4.6 HMC Power Consumption and Thermal Analysis 37
3.5 Chapter Conclusions 40

4. NEAR DATA COMPUTING USING COMMODITY MEMORY 41

4.1 Background 41
4.1.1 MapReduce Workloads 41
4.1.2 An Optimized Baseline for MapReduce 42

4.2 Near Data Computing Architectures 43
4.2.1 NDC with HMCs 43
4.2.2 NDC with Modules 44
4.2.2.1 NDC-Module Cards 44
4.2.2.2 LPDDR2 x32 DRAM Chips 45
4.2.2.3 NDC-Module Motherboard Organization 45
4.3 Evaluation 46
4.3.1 Evaluated Systems 46
4.3.2 Methodology 46
4.3.3 Workloads 47
4.4 Performance Results 48
4.4.1 Bandwidth 48
4.4.2 MapReduce Performance 49
4.4.3 Energy Consumption 50
4.4.4 Cost 51
4.5 Chapter Conclusions 52

5. NEAR DATA COMPUTING AND FIXED FUNCTION ACCELERATORS 53

5.1 Map Phase Execution Breakdown 53
5.2 Hardware Sorting Strategies 55
5.2.1 Sorting on Keys 55
5.2.2 Hardware Sort Algorithms 55
5.2.3 Hardware Merge Sort 56
5.2.3.1 Two-Input Merge Sorter 57
5.2.3.2 Eight-Input Merge Sorter 58
5.2.4 Hardware Bitonic Sort 59
5.2.4.1 Reliance on Other Sorting Mechanisms 60
5.3 Hardware Sort Requirements 60
5.3.1 Two-Input Merge Sorter 60
5.3.2 Eight-Input Merge Sorter 61
5.3.3 Bitonic Sort with Range Partitioner 61
5.3.4 Bitonic Sort with Merge Sorter 62
5.3.5 Summary of Hardware Sorting Unit Characteristics 62
5.4 Integrating Hardware Sorting into NDC 64
5.4.1 Hardware Changes 64
5.4.2 Software Changes 65
5.5 Evaluating Hardware Sorting in NDC 66
5.5.1 Methodology 66
5.5.2 Performance Evaluation 67
5.5.2.1 Sorter Data Transfer Requirements 67
5.5.2.2 Sorting Latency 68
5.5.2.3 Map Task Latency 69
5.5.2.4 Sorting Compute Energy 70
5.5.2.5 Map Task Energy 71
5.5.2.6 Sorting Accelerators with NDC-Module 73
5.6 Chapter Conclusions 75

6. CONCLUSION AND FUTURE WORK 77

6.1 Contributions 77
6.1.1 Near Data Computing with 3D-Stacked Memory+Logic Devices 77
6.1.2 Near Data Computing Using Commodity Memory 78
6.1.3 Near Data Computing and Fixed Function Accelerators 78
6.1.4 Contributions Discussion 79
6.2 Future Work 80
6.2.1 Near Data Computing and Iterative MapReduce 80
6.2.2 Accelerating General Purpose Map and Reduce Functions with Reconfigurable Pipes 80
6.2.3 Near Data Computing and General Database Acceleration 81

REFERENCES 82

LIST OF TABLES

3.1 Memory technology comparison. 20

3.2 Energy Efficient Core (EECore) and baseline systems. 23

3.3 Parameters for evaluated Out-of-Order, EECore, and NDC systems. . . 28

4.1 Parameters for evaluated EECore, NDC-HMC, and NDC-Module systems. 46

5.1 Parameters for Q100 units, used as building blocks to construct hardware sorting accelerators. 61

5.2 Hardware sort accelerator parameters. 63

5.3 Output characteristics of each workload, and the number of sorting scans through memory needed for MergeSort2, MergeSort8, BitonicSort+RangePartition, and BitonicSort+MergeSort8. 67

ACKNOWLEDGEMENTS

In 2008, I graduated from the University of Utah with a bachelor's degree in computer science, and I was ready to go off to New York City to attend law school, despite everyone I know telling me the world didn't need any more lawyers. While I had learned a lot in my undergraduate career, and even had the chance to engage in some computer architecture research with Rajeev Balasubramonian and his students, I was sure that the right move for me was to go to law school. I didn't have another plan.

That changed when, during the week of graduation, my best friend, Melina Vaughn, decided she wanted to be my girlfriend. I suddenly had a reason to not move to New York. However, I didn't have a plan for what to do if I stayed in Salt Lake City. That's when my undergraduate research advisor, Rajeev, said he would vouch for me, and "sneak" me into graduate school at the School of Computing, despite having never applied. So now I had a compelling reason to stay in Salt Lake City, and a great opportunity for something to do when I stayed.

I share this story because these two people (Melina and Rajeev) had the greatest, most direct influence on my attending graduate school in the first place, and they were also my greatest support throughout the experience. Fortunately, both sides of this story have happy endings. I ended up marrying Melina in October 2008, and now in 2014 I have finally graduated with my Ph.D.

My committee has helped to keep me on track over the years, and filled in the gaps in my knowledge. Over the years, many discussions with Al and Erik have involved references to electrical phenomenon that I frankly didn't understand. I always just hoped that I would be able to piece things together from context, and sometimes I could, but it was always very embarrassing when I'd have to have to admit I had no clue what they were talking about, and that I hadn't been able to follow the conversation for several minutes. Viji was just a voice on a phone for years before we finally met in person, but even over the phone, she often provided much needed doses of reality to temper some of our more outlandish and unrealistic research ideas. Feifei in particular did a lot to teach me about big data workloads, and especially MapReduce, and gave me a lot of the background that made this work possible. I never actually took a class on big data; I only had Feifei to learn from directly.

Also during my years of graduate school, I spent some time interning with Intel Labs, working closely with Chris Wilkerson and Zeshan Chishti. While the work I did with them is totally orthogonal to the contents of this dissertation, they both were great resources and helped to broaden the horizons of my computer architecture knowledge.

The Utah Arch lab has been a fun place to learn and work together. It has had a wild cast over the years, and just when one particularly zany character graduates, and you think the lab will never be as fun again, another shows up and eventually reveals that they're just as crazy. Despite all this, we got a lot of good work done together, even if my infinite write queue idea never quite came together like I wanted.

It has been a tremendous boon to me to attend graduate school in the city where I grew up, because I've been able to see my family so often. Weekly Saturday lunches and holidays spent with family have provided a stability and source of joy that have helped me keep going. Computers are nice and interesting, but family is way better.

Finally, I want to thank Ann and Karen at the School of Computing for making so many of my problems magically disappear over the years.

CHAPTER 1

INTRODUCTION

1.1 Emerging Trends

Processing in Memory (PIM) is an old idea that has been well studied in recent decades, but it has never caught on commercially. Cost considerations are a major contributor to this, due to the requirement that both compute and memory circuits be integrated into the same chip using the same manufacturing technology. This required sacrificing either the density of the memory or complexity of the computation logic. Either way, compromises were made, and the resulting PIM chips were held back from gaining commercial acceptance.

However, in recent years, several technological trends are converging that make the idea of PIM, now often called Near Data Computing (NDC), or Near Data Processing (NDP), an attractive option once again. New classes of workloads have emerged that can greatly benefit from the NDC model, and new manufacturing techniques are emerging that eliminate the need for the costly compromises that held back NDC in the past. We will next examine some of these trends and how they make an argument for NDC.

1.1.1 Big Data

Big Data refers to a class of applications that deal with data sets so large that managing the data itself is one of the primary challenges. These data sets are often so large that they must reside on disk, but customers continue to demand ever larger DRAM-based main memory systems to avoid as many slow disk accesses as possible.

Main memories grow as more ranks are added per memory module, and more memory modules are added per channel, enabled by using buffer and register chips. This increase in main memory capacity is not accompanied by increases in main memory bandwidth or reductions in memory latency.

As a result, more and more data must be transmitted between the main memory and host memory controllers, using the same memory channels operating at the same frequency as before. Adding more memory capacity does not add more memory bandwidth. This is one major advantage of NDC, where adding more memory capacity to the system also adds more aggregate memory bandwidth and more compute capability.

1.1.2 MapReduce

MapReduce [1] is a popular framework for programming and executing data parallel Big Data applications, and it is new since the last time NDC was seriously investigated. From the programmer's perspective, all they do is write Map and Reduce functions, and the MapReduce runtime, such as Hadoop [2], transparently takes care of harnessing the compute capability of an entire cluster to complete the task.

MapReduce workloads are parallel on "records," which could obviously refer to database records. However, other types of non-database workloads can be expressed in terms of MapReduce. The Map function is some operation that you want to apply to all of your records. The Reduce function aggregates the results from the Map function. Not all parallel applications are an obvious fit for MapReduce, but any data parallel problem that can be expressed in terms of records, a Map function, and a Reduce function can be made to work.

1.1.3 In-Memory Databases

While current incarnations of Big Data systems rely on disks for most data accesses, there is a growing trend towards placing a large fraction of the data in memory, e.g., Memcached [3], RAMCloud [4], and Spark [5]. The RAMCloud project shows that if a workload is limited more by data bandwidth than by capacity, it is better to store the data in a distributed memory platform than in a distributed HDD/SSD platform.

There are therefore many big-data workloads that exhibit lower cost and higher performance with in-memory storage [4]. This is also evident in the commercial world, e.g., SAS in-memory analytics [6], SAP HANA in-memory computing and in-memory database platform [7], and BerkeleyDB [8] (used in numerous embedded applications that require a key-value-like storage engine).

Let us consider SAP HANA as a concrete example. It employs a cluster of commodity machines as its underlying storage and computation engine, and it relies on the collective DRAM memory space provided by all nodes in the cluster to store large data entirely in memory. Each node can provide tera-bytes of memory, and collectively, they deliver an in-memory storage space that can hold up to hundreds of terabytes of data, depending on the size of the cluster [9, 10].

Spark in particular is an extension of MapReduce that aims to store as much of the data set in memory as possible, and avoid costly disk accesses that are the norm in conventional MapReduce frameworks like Hadoop. Spark is built on the idea of Resilient Distributed Datasets (RDDs) that express some transformation from the original input data set, or from another RDD, and can be explicitly cached in memory. This gives the programmer some ability to control the main memory-residence of the data set, in order to maximize MapReduce performance.

1.1.4 Commodity Servers

A high end Intel Xeon server today may make use of four CPU sockets, each populated with a 2.8 GHz, 15-core high performance out-of-order CPU [11]. Each of these CPUs is connected to four channels of memory, providing up to 85 GB/s of total memory bandwidth per socket.

This is a very high end example of the kind of system that might be used to execute MapReduce workloads. It focuses primarily on single-threaded performance, using energy-inefficient out-of-order cores running at very high clock speeds, and supporting relatively few hardware threads. Each thread of a MapReduce task may be executed very quickly, but this kind of system lacks the level of parallelism that is inherent in many MapReduce workloads.

1.1.5 3D-Stacked Manufacturing

3D stacking technology has made it a lot easier to tightly integrate compute and memory, enabling a new generation of processing in memory, or near data processing. In 3D stacking, multiple dies are manufactured independently, and then bonded together to form one chip. These dies may even be manufactured on different process nodes or manufacturing technology, as long as they do have an interface to connect with one another using Through-Silicon Vias (TSVs).

This offers an opportunity to build systems for Big Data that do not resemble the commodity servers on which these workloads typically run, and are better tailored to their particular needs. It enables NDC, which can bring compute and memory closer together, and helps scarce resources like memory bandwidth scale as more and more memory is added to the Big Data system.

1.2 Dissertation Overview

We can see from these emerging trends that Big Data applications will require more and more main memory, and that 3D-stacked Near Data Computing is one way that memory bandwidth and compute resources can scale with increased memory capacity in a way that commodity servers are incapable of doing. In this dissertation, we look at how NDC can enable higher performance and better energy efficiency for in-memory MapReduce workloads. First, in Chapter 3 we investigate what it would take to build an NDC system, and how MapReduce workloads could run on it. We compare this NDC system to a baseline with few out-of-order cores, and another baseline with many in-order cores. Both of these baselines are equipped with HMC main memory systems, capable of delivering very high amounts of memory bandwidth. The way these baseline systems differ from each other and from the NDC system is in the number and type of processing cores they have, as well as how those cores access memory. Second, in Chapter 4 we relax our definition of NDC to include tightly coupled CPUs and DRAM that are in separate packages connected via conventional memory buses. In this design, we retain the goal of NDC to scale memory bandwidth as memory capacity increases by coupling each DRAM chip with a dedicated processor. Finally, in Chapter 5 we look at the potential for introducing fixed-function hardware accelerators into the MapReduce workflow to improve sort performance. We find that such accelerators offer such high performance that they require NDC to deliver the memory bandwidth they require.

1.2.1 Thesis Statement

The data-parallel nature of MapReduce makes it a poor fit for the commodity server clusters it typically runs on. Near data computing architectures that focus on compute throughput and memory bandwidth will offer the greatest performance and energy characteristics for these workloads. Further, while general programmability is an intrinsic requirement of the MapReduce model, there are opportunities for fixed-function accelerators to have a large impact on MapReduce performance and energy consumption.

1.2.2 Near Data Computing with 3D-Stacked Memory+Logic Devices

While Processing in Memory has been investigated for decades, it has not been embraced commercially. A number of emerging technologies have renewed interest in this topic. In particular, the emergence of 3D stacking and the imminent release of Micron's Hybrid Memory Cube device have made it more practical to move computation near memory. However, the literature is missing a detailed analysis of a killer application that can leverage a Near Data Computing (NDC) architecture. This work focuses on in-memory MapReduce workloads that are commercially important and are especially suitable for NDC because of their embarrassing parallelism and largely localized memory accesses. The NDC architecture incorporates several simple processing cores on a separate, non-memory die in a 3D-stacked memory package. These cores can perform Map operations with efficient memory access and without hitting the bandwidth wall. This work describes and evaluates a number of key elements necessary in realizing efficient NDC operation: (i) low-EPI cores, (ii) long daisy chains of memory devices, (iii) the dynamic activation of cores and SerDes links. Compared to a baseline that is heavily optimized for MapReduce execution, the NDC design yields up to 15X reduction in execution time and 18X reduction in system energy.

1.2.3 Near Data Computing Using Commodity Memory

The cost of HMC-style memory may prove to be an insurmountable barrier to the adoption of Near Data Computing (NDC) as previous described. We instead consider a variation on the theme of NDC that includes commodity processors and DRAM chips, rather than specialized 3D-stacked all-in-one chips. While the goal of scaling memory bandwidth with memory capacity is the same, the implementation is different. Instead of each core in an NDC device having a dedicated vault of memory within the HMC, a small number of low-power cores will share a conventional data bus connected to a single LPDDR2 x32 DRAM chip. This pattern is repeated for all the DRAM chips in the system, so that each DRAM chip may offer its entire memory bandwidth simultaneously to work toward MapReduce execution. This is in contrast to a conventional commodity server where all of the ranks on a channel of memory must take turns using the data bus, forcing a large fraction of all DRAM chips to sit idle at all times. We find that this technique again improves performance compared to the optimized MapReduce server baseline, but it does not perform as well as an HMC-based NDC system.

1.2.4 Near Data Computing and Fixed Function Accelerators

A large fraction of MapReduce execution time is spent processing the Map phase, and a large fraction of Map phase execution time is spent sorting the intermediate key-value pairs generated by applying the Map function to the input data split. This work focuses on accelerating the sorting process within the Map phase using fixed-function hardware accelerators. Sorting accelerators can achieve high performance and low power use because they lack the overheads of sorting implementations on general purpose hardware such as instruction fetch and decode, RISC-style execution where each instruction must be executed in a different cycle, and loop overheads. We find that sorting accelerators are a good match for HMC-based Near Data Computing (NDC) because their sorting performance is so high that it quickly saturates the memory bandwidth available in the commodity memory-based implementation of NDC. This increased sorting performance and low power requirement of fixed-function hardware lead to very high Map phase performance and task energy use. These large advantages could make a compelling case for HMC-based NDC, despite the high cost.

CHAPTER 2

BACKGROUND

In this chapter, we will look at the MapReduce framework, 3D stacked manufacturing, processing in memory, and hardware accelerators for Big Data workloads. Each of these plays a large role in subsequent chapters, where we design and evaluate systems that use principles of processing in memory, 3D stacking, and hardware acceleration to improve the performance of MapReduce workloads.

2.1 MapReduce

It is becoming increasingly common for companies such as Google and Facebook to maintain datasets of several terabytes. A popular computation model for processing large datasets in parallel is MapReduce [1] deployed on a cluster of commodity machines. In MapReduce, a user need only define a map and reduce function and transparently gains the compute power of the entire cluster in addition to fault tolerance offered by MapReduce.

MapReduce workloads are performed by two agents, Mappers which execute the Map phase, and Reducers which execute the Reduce phase, as seen in Figure 2.1. In MapReduce, an input data set is divided into input “splits.” Each split is operated on by a different Mapper. The more splits that can be processed in parallel, the higher the performance of the MapReduce cluster. We next describe the work done in the Map and Reduce phases.

2.1.1 Map

In the Map phase, each Mapper scans through the records in its input data split, and produces an intermediate output in the form of key-value pairs. This output is then partitioned and portions of the output from each Mapper task is sent to each Reducer task.

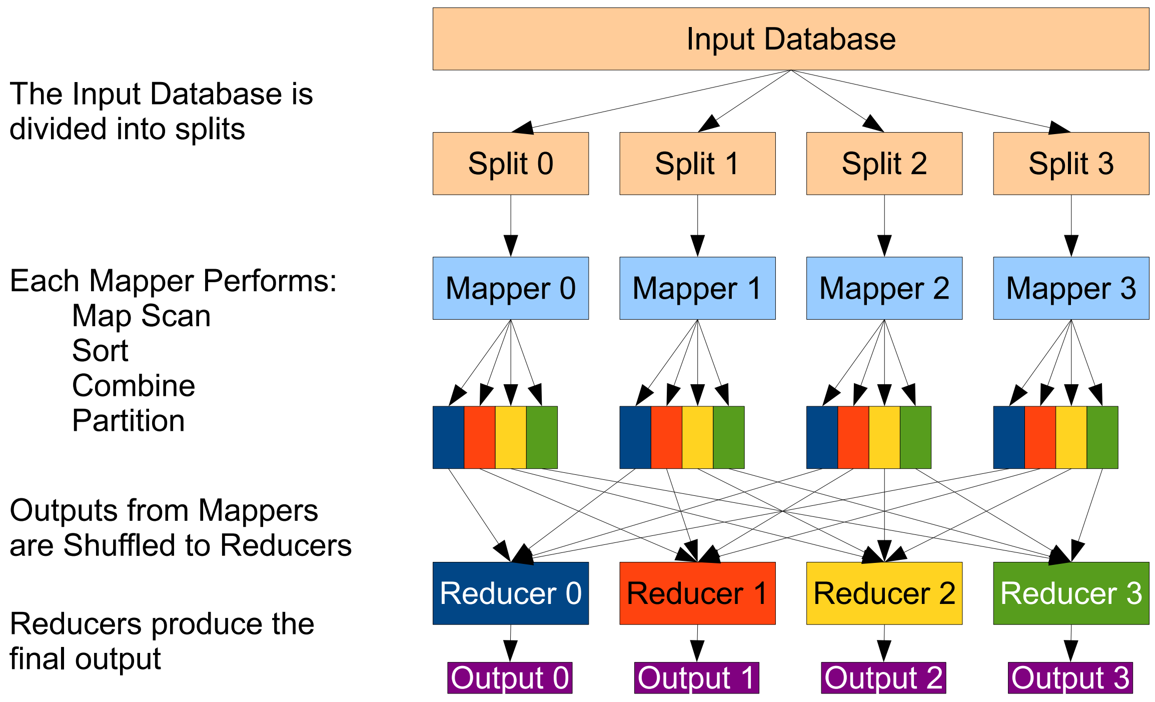

Input Database
The Input Database is divided into splits
Split 0
Split 1
Split 2
Split 3
Each Mapper Performs: Map Scan Sort Combine Partition
Mapper 0
Mapper 1
Mapper 2
Mapper 3
Outputs from Mappers are Shuffled to Reducers
Reducer 0
Reducer 1
Reducer 2
Reducer 3
Reducers produce the final output
Output 0
Output 1
Output 2
Output 3

Figure 2.1. How data move through a MapReduce workload.

2.1.1.1 Map Scan

The Map Scan phase reads in every record in the input data split, and applies the user-supplied Map function to it, producing a key-value pair as output. One common function that is used in MapReduce applications is the identity function, where some key is created based on the input record, and the value is the number one.

This is useful for aggregation queries that will ultimately count the number of instances of each key, and the value represents how many times that key has been seen. In this case, the Map Scan produces many key-value pairs, some of which may have duplicate keys, and assigns all of them the value of one. Later MapReduce phases will consolidate the key-value pairs based on their keys, and sum up their values.

2.1.1.2 Sort

One common goal of MapReduce applications is to aggregate key-value pairs with identical keys. This is easily accomplished when key-value pairs have been sorted on their keys, because identical keys will appear consecutively in sorted order. Any sorting method can be used to accomplish this sorting, and in this work we primarily use the merge sort algorithm.

In Section 5.1, we find that the Sort phase takes up the majority of Map phase time for in-memory MapReduce workloads.

2.1.1.3 Combine

The Combine phase can be viewed as a miniature, local Reduce. It commonly performs the same aggregation that the Reduce phase does, but it does it only on the intermediate output from the current, local Mapper task.

Using again the example of aggregation, a combine phase might scan through the sorted key-value pairs from the Sort phase, and count the number of occurences of each key (which is easy to do if the data are in sorted order). It will then emit a key-value pair for each key it sees along with a value that represents the number of instances of that key.

The degree to which this decreases the volume of data to be processed by the Reducer depends on the particular keys being combined. In the workloads we consider in later chapters, the Combine phase is very successful at decreasing the size of the output of the Map phase.

2.1.1.4 Partition

The intermediate outputs from the Map phase need to be partitioned according to their keys so that all of the key-value pairs with the same key can ultimately be processed by the same Reducer. The method of partitioning depends on the nature of the particular MapReduce application. For many workloads it is sufficient to use a simple deterministic hash partitioning function to assign a key-value pair to a Reducer. In other domains, it could make more sense to perform a range partitioning function on the key-value pairs to assign them to Reducers.

2.1.2 Shuffle and Sort

Between Map and Reduce phases, data are moved from every Mapper to every Reducer, so that key-value pairs with the same keys will all be processed by the same Reducers. For conventional commodity clusters this requires traversing NICs and routers to move data from Mapper to Reducer, and can arise as a significant bottleneck to overall performance. However, in this work, we focus on in-memory MapReduce applications where all the data can be stored within a single box, and no network communication is required. Therefore, this is not a major bottleneck in this work, and instead we focus on improving the compute performance and memory bandwidth characteristics of a single system.

2.1.3 Reduce

The Reduce phase performs a similar function to the Combine phase, except it operates on key-value pairs from all Mappers in the cluster. Because of the partitioning that was done earlier, all instances of a particular key are guaranteed to be found within a single Reducer. Each Reducer processes these key-value pairs and produces a final output. The final output is all of the outputs from each Reducer task considered together as one whole.

2.2 3D Stacking

A number of recent papers have employed 3D stacking of various memory chips on a processor chip (for example, [12–18]). 3D stacking is a manufacturing technique that enables the integration of multiple silicon die in a vertical fashion into a single package. Industrial 3D memory prototypes and products include those from Samsung [19, 20], Elpida [21, 22], Tezzaron [23], and Micron [24, 25]. This allows a single chip to not only have a single silicon layer and multiple metal layers, but multiple silicon layers, each with transistors and gates, along with a full complement of metal layers for each silicon layer.

2.2.1 Through-Silicon Vias

Through-Silicon Vias (TSVs) are a way to connect multiple silicon dies on top of each other that have been manufactured independently of one another [21], as seen in Figure 2.2. TSVs have their own layout constraints, similar to regular metal layer vias but with even greater pitches and area overheads. While there are definite limits to the number of TSVs that can be integrated into a single die, the number of TSV

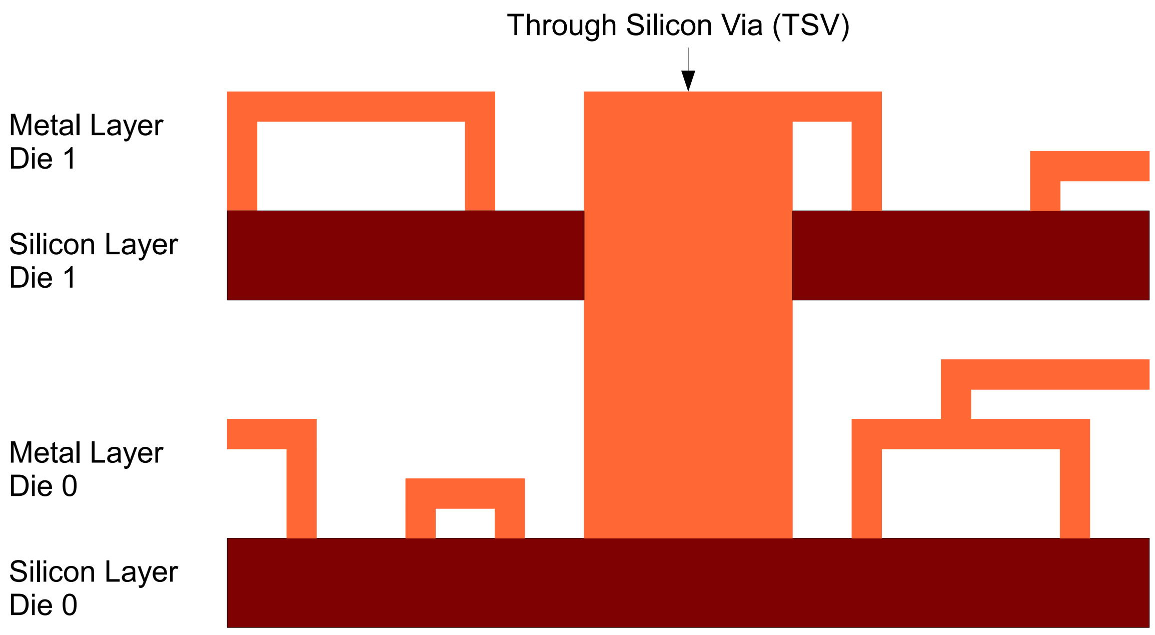


Figure 2.2. This cross-sectional view shows that Through-Silicon Vias allow for signals to be passed through silicon chips that are stacked on top of and bonded to one another.

connections between dies that are vertically integrated is much greater than off-chip connections, or connections within a multichip module (MCM). This allows for much higher bandwidth between layers, at lower energy cost, because there is no need for energy-hungry off-chip drivers.

2.2.2 Hybrid Memory Cube

The Hybrid Memory Cube (HMC) is a product developed by Micron Technology [26] that uses 3D Stacking to integrate several DRAM layers with a single logic layer, connected by TSVs, as seen in Figure 2.3. The HMC is designed for very high bandwidth and low latency, so the internal structure of the DRAM layers departs somewhat from conventional commodity DRAM. The logic layer of the HMC contains high-speed SerDes links that allow for very high bandwidth between an HMC device and its host.

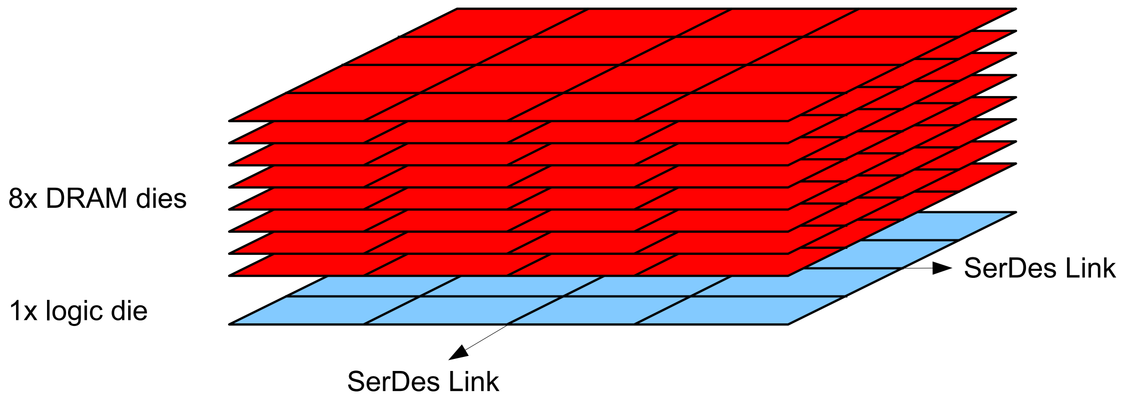


Figure 2.3. A Hybrid Memory Cube with eight DRAM layers and one logic layer.

2.2.2.1 DRAM Organization

Rather than having 8 banks per DRAM layer, as would be common in a DDR3 chip [27], a single HMC DRAM layer has 32 banks, organized into 16 vaults. This increases the number of independent memory transactions on which a single die can operate by a factor of 4x. The degree of parallelization is increased by the fact that an HMC device integrates several DRAM dies. If an HMC has 8 DRAM layers, and each DRAM layer has 32 banks, then the total number of banks in an HMC would be 256, an increase in potential bank-level parallelism over DDR3 by a factor of 32.

Furthermore, each vault acts as an independent channel of memory, so there is also vault-level parallelism. For example, if each vault has a data width of 32 bits, and each data bus TSV runs at 1600 Mbps, then each vault is able to supply 6.4 GB/s of bandwidth. Since each vault can act independently, a single HMC device with 16 vaults can offer up to 102.4 GB/s of bandwidth.

2.2.2.2 Logic Layer

The logic layer of the HMC serves several functions. First, it is where the per-vault memory controllers reside. Second, it has several high-speed SerDes links for communicating off-chip. Third, it has routing capability between the SerDes links and vaults, and makes sure memory requrests are handled by the appropriate vault.

A host memory controller for interfacing wth HMC does not deal directly with the DRAM in terms of row activates, column reads and writes, and precharge commands.

Instead, the protocol for performing memory reads and writes is packet-based, and the memory controller just builds and sends packets to the HMC, and the HMC does likewise in response.

It is possible to build networks or daisy chains out of HMC devices, as seen in Figure 2.4. This makes it possible to build large main memories out of HMC devices. While each HMC device is capable of independently providing up to 102.4 GB/s of bandwidth, as per our earlier example, there is a practical limit on how much memory bandwidth the host memory control can see. This is limited by the speed and width of the SerDes links directly connecting the host memory controller to the first HMC in a daisy chain or network. Extending a daisy chain, for example, only increases capacity, it does not increase bandwidth at the host memory controller.

2.3 Processing in Memory

Processing in Memory (PIM) is the idea of tightly integrating some compute capability with some memory. PIM was originally proposed with a 2D implementation, but recent advances in 3D stacking manufacturing allow a more practical implementation, as seen in Figure 2.5. A number of PIM projects are summarized below. While a few

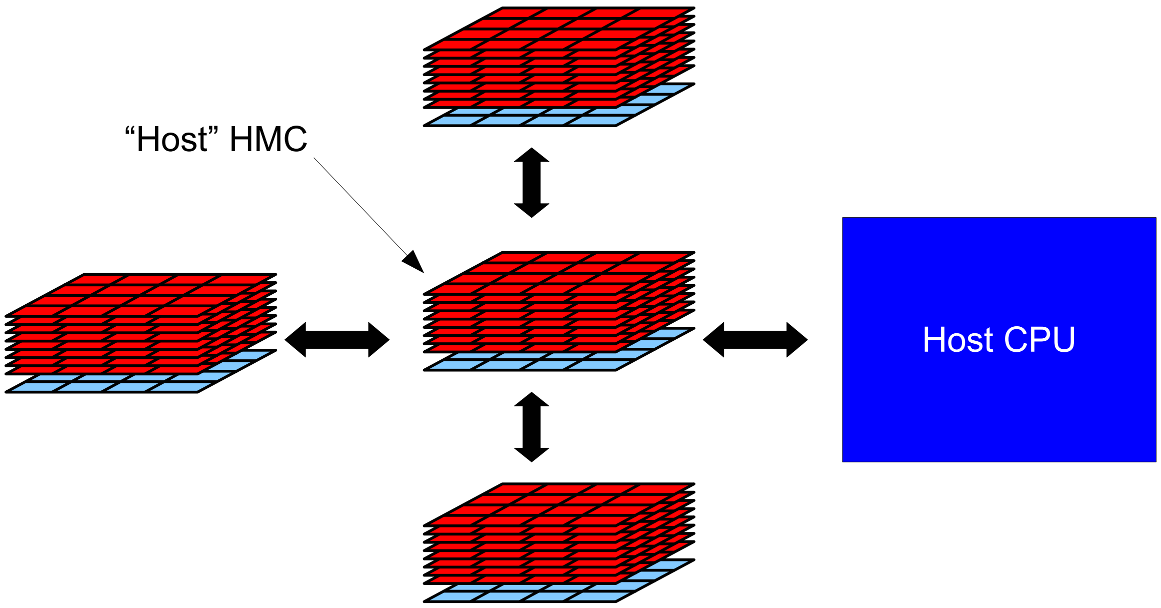


Figure 2.4. A network of HMC devices attached to a host CPU. The HMC attached to the host CPU acts as a host for the HMCs attached to it.

have examined database workloads, none have leveraged the MapReduce framework to design the application and map tasks automatically to memory partitions. Many of these prior works have support for thread migration among PIM nodes and require a network among PIM nodes for data transfer and thread migration.

2.3.1 DIVA

The DIVA project from USC and Notre Dame connected PIM memories to an external host processor and a memory-to-memory interconnect [28, 29]. The PIM chip is therefore akin to a co-processor. A single PIM chip dedicates 60% of its area to memory and 40% to logic. They employ simple in-order 5-stage RISC pipelines with an integer ALU and a 256-wide SIMD ALU. The PIM chip also has instruction caches and register files. The memory space is partitioned into regions that are only touched by the host CPU, or only touched by PIM, or both. The PIM chip maintains a TLB-like structure to store virtual-to-physical mappings for the memory space accessible by PIM. While a primary focus of the project was scientific workloads, the project also evaluated a natural-join database operation [28].

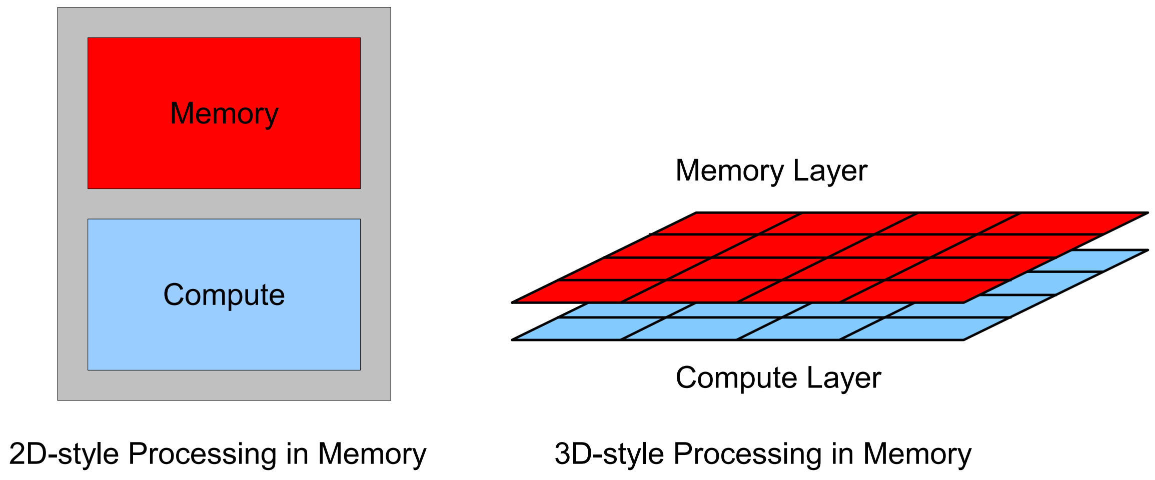


Figure 2.5. 2D-style PIM was forced to use the same manufacturing process for both compute and memory devices on the same die. 3D-style PIM allows for both compute and memory components to be manufactured independently with optimized manufacturing technologies.

2.3.2 Vector IRAM

Berkeley's Intelligent RAM project focused on mobile multimedia devices [30, 31], characterized by small memory footprints and many vector operations. Vector IRAM chips were designed to serve as stand-alone units. The chip consists of eight embedded DRAM banks, four vector processor lanes, and a scalar core. Early IRAM position papers focus on media workloads, but mention its eventual use in servers and supercomputers to perform data mining [32].

2.3.3 PIM-Lite

Like the DIVA project, Notre Dame's PIM-Lite [33] chip is intended to serve as a co-processor. Each card can accommodate 16 PIM chips and each chip has 4 SIMD processing units. The project acknowledges a doubling in the cost of a memory chip, but projects a 4x increase in performance for irregular scientific applications. The project also examines a variety of data-intensive irregular applications, including a query operation on a database with an R-Tree indexing scheme [34].

2.3.4 FlexRAM

The FlexRAM project [35] designs PIM chips that can be placed on DIMMs and can be flexibly used as PIM or regular DRAM. Each FlexRAM chip incorporates many simple cores on roughly one-third of the chip area. The project examines a wide variety of multimedia and datamining workloads including tree generation, tree deployment, and database query processing [36]. Follow-on work builds compiler support to automatically map applications to a PIM model [37].

2.3.5 Gilgamesh

The Gilgamesh chip [38] extends PIM functionalities with task/data virtualization and adaptive resource management. Many PIM chips can form a network and they may or may not be controlled by a host CPU. Wide SIMD ALUs are associated with each row buffer. The workloads include scientific vector codes.

2.3.6 Mercury

The Mercury architecture [39] uses a 3D-stacked PIM architecture to increase throughput for Memcached workloads. A single Mercury system consists of dozens of individual nodes on a motherboard. Each node is a 3D-integrated system, including 4 GB DRAM, 2 ARM Cortex A7 CPUs, and a NIC. This architecture is proposed as a way to increase processing and memory density (in terms of rack space in a data center) for Memcached workloads.

2.3.7 Other Related PIM Work

A paper from Sandia examines the use of PIMs within network interface cards to accelerate MPI transactions [40]. Some early PIM work by the Supercomputing Research Center focused on DNA sequence match, image processing, and matrix algorithms [41]. The Smart Memories project [42] implements a reconfigurable tile with processing, memory, and interconnect, that can be used to implement a variety of processor architectures. Saulsbury et al. [43] combine a 5-stage pipeline with a DRAM chip for a 10% area overhead and evaluate behavior for SPEC and SPLASH benchmarks. The Active Pages [44] architecture integrates reconfigurable logic on DRAM chips for regular applications, including unindexed data search.

2.4 Hardware Accelerators for Big Data Applications

Recently, several hardware accelerators have been proposed to address performance issues in the area of Big Data applications. In the space of Big Data, there are two large classes of applications, those that are query-based, and those that are transaction-based. Query-based applications must examine every element in a large data set, and typically produce results answering questions about trends within the data set. MapReduce is a common example of a query-based Big Data application, which is the kind of workload that we target in this work.

Transaction-based applications, on the other hand, do not touch every piece of data in a large data set. Rather, they use indexing structures to target one specific piece of data, and then read, modify, add, or delete a single item in the larger data set. Memcached is a common example of a transaction-based Big Data application. A single transaction consists of a user asking for the data associated with a single key, and the Memcached system returns that item, and only that item, not touching any other items in the database extraneously.

2.4.1 Thin Servers with Smart Pipes

Thin Servers with Smart Pipes (TSSP) [45] is a hardware accelerator design proposed by Lim et al. that targets Memcached. They identify several bottlenecks that are holding back conventional systems from high performance, and they look at both servers with high-end out-of-order CPUs and low-end in-order CPUs. They find that even though the code footprint of Memcached is very small, in practice it makes many trips through the TCP/IP stack, kernel space, and other library code.

They implement a TSSP prototype using an FPGA that accelerates the processing of ethernet packets, and performs the lookups that Memcached requires. By offloading this work from the host CPU, this accelerator improves performance per watt by a factor of 16x.

2.4.2 HARP

Range partitioning was identified by Wu et al. [46] as a major performance bottleneck in many query-based database workloads, especially as it is a vital component for database join operations. Although hashing-based partitioning is trivial to compute, range partitioning is much more computationally intensive, and they found that large amounts of system memory bandwidth was going unused during range partitioning phases of program execution. Their goal was to accelerate this computation to use more of the available bandwidth.

They designed a hardware accelerator, called HARP for "Hardware Accelerator for Range Partitioning," that is able to partition one record per cycle. They wrote verilog code, and went through the place and route workflow to quantify the area, power, and performance characteristics of their HARP accelerator. Their HARP accelerator can partition an input data set at a rate equal to 16 software partitioning threads running on a high-end server.

2.4.3 Q100

Building on their work with HARP, Wu et al. next turned their focus to accelerating all parts of an SQL-style query with their Q100 Database Processing Unit (DPU) [47]. Rather than just considering a range partitioning unit, they now turn their attention to many different acceleration units, including an Aggregator, an ALU, a Boolean Generator, a Column Filter, a Joiner, a Partitioner, a Bitonic Sorter, and Append, Column Select, Concatenation, and Stitching units.

All of these units in concert can process many SQL queries (with some limited exceptions). A database query can be translated into a query plan, in the form of the actions and interactions of all these hardware units, and then execution of that plan can be performed with very high throughput.

They consider input databases both with their initial size, and a version of those databases that are 100x larger. As the database size grows, the performance improvement of using Q100 versus a conventional system diminishes. They do not explore the bounds of this waning performance advantage, or if their Q100 design ever drops behind in performance compared to a conventional system for very large input database sizes. One weakness of the Q100 work is that they only consider the operation of a single DPU operating in isolation, and make no attempt to take advantage of the data parallelism that is inherent in many database operations.

CHAPTER 3

NEAR DATA COMPUTING WITH 3D-STACKED MEMORY+LOGIC DEVICES

There is great interest in designing architectures that are customized for emerging big-data workloads. For example, a recent paper [45] designs a custom core and NIC for Memcached. In this work, we make a similar attempt for in-memory MapReduce workloads. We take advantage of emerging 3D-stacked memory+logic devices (such as Micron's Hybrid Memory Cube or HMC [48]) to implement a Near Data Computing (NDC) architecture, which is a realizable incarnation of processing in memory (PIM). A single board is designed to accommodate several daisy chained HMC-like devices, each of which includes a few low-power cores that have highly efficient access to a few giga-bytes of data on the 3D stack. MapReduce applications are embarrassingly parallel and exhibit highly localized memory access patterns, and are therefore very good fits for NDC.

While Memcached has the same behavior on every invocation and can benefit from customization [45], MapReduce functions can take several forms and therefore require programmability. We show that efficiency and cost are optimized by using low Energy Per Instruction general-purpose cores, by implementing long daisy chains of memory devices, and by dynamically activating cores and off-chip SerDes links. Our work is the first thorough quantitative study analyzing the effect of HMC-like devices for in-memory Map-Reduce workloads. We lay the foundation for future studies that can further explore the NDC architecture and apply it to other big-data workloads that have high degrees of parallelism and memory locality.

3.1 Memory System Background

3.1.1 Moving from DDR3 to HMC

In a conventional memory system, a memory controller on the processor is connected to dual in-line memory modules (DIMMs) via an off-chip electrical DDR3 memory channel (bus). Modern processors have as many as four memory controllers and four DDR3 memory channels [49]. Processor pin counts have neared scaling limits [50]. Efforts to continually boost processor pin bandwidth lead to higher power consumption and limit per-pin memory capacity, thus it is hard to simultaneously support higher memory capacity and memory bandwidth.

Recently, Micron has announced the imminent release of its Hybrid Memory Cube (HMC) [25]. The HMC uses 3D die-stacking to implement multiple DRAM dies and an interface logic chip on the same package. TSVs are used to ship data from the DRAM dies to the logic chip. The logic chip implements high-speed signaling circuits so it can interface with a processor chip through fast, narrow links.

3.1.2 Analyzing an HMC-Based Design

In Table 3.1, we provide a comparison between DDR3, DDR4, and HMC-style baseline designs, in terms of power, bandwidth, and pin count. This comparison is based on data and assumptions provided by Micron [26, 48].

HMC is optimized for high-bandwidth operation and targets workloads that are bandwidth-limited. HMC has better bandwidth-per-pin, and bandwidth-per-watt characteristics than either DDR3 or DDR4. We will later show that the MapReduce applications we consider here are indeed bandwidth-limited, and will therefore best run on systems that maximize bandwidth for a given pin and power budget.

Table 3.1. Memory technology comparison.

	Pins	Bandwidth	Power
DDR3	143	12.8 GB/s	6.2 W
DDR4	148	25.6 GB/s	8.4 W
HMC	128	80.0 GB/s	13.4 W

3.2 Near Data Computing Architecture

3.2.1 High Performance Baseline

A Micron study [26] shows that energy per bit for HMC access is measured at 10.48 pJ, of which 3.7 pJ is in the DRAM layers and 6.78 pJ is in the logic layer. If we assume an HMC device with 4 links that operate at their peak bandwidth of 160 GB/s, the HMC and its links would consume a total of 13.4 W. About 43% of this power is in the SerDes circuits used for high-speed signaling [26, 51]. In short, relative to DDR3/DDR4 devices, the HMC design is paying a steep power penalty for its superior bandwidth. Also note that SerDes links cannot be easily powered down because of their long wake-up times. So the HMC will dissipate at least 6 W even when idle.

We begin by considering a server where a CPU is attached to 4 HMC devices with 8 total links. Each HMC has a capacity of 4 GB (8 DRAM layers each with 4 Gb capacity). This system has a memory bandwidth of 320 GB/s (40 GB/s per link) and a total memory capacity of 16 GB. Depending on the application, the memory capacity wall may be encountered before the memory bandwidth wall.

Memory capacity on the board can be increased by using a few links on an HMC to connect to other HMCs. In this chapter, we restrict ourselves to a daisy chain topology to construct an HMC network. Daisy chains are simple and have been used in other memory organizations, such as the FB-DIMM. We assume that the processor uses its 8 links to connect to 4 HMCs (2 links per HMC), and each HMC connects 2 of its links to the next HMC in the chain (as seen in Figure 3.1b). While daisy chaining increases the latency and power overhead for every memory access, it is a more power-efficient approach than increasing the number of system boards.

For power-efficient execution of embarrassingly-parallel workloads like MapReduce, it is best to use as large a number of low energy-per-instruction (EPI) cores as possible. This will maximize the number of instructions that are executed per joule, and will also maximize the number of instructions executed per unit time, within a given power budget. According to the analysis of Azizi et al. [52], at low performance levels, the lowest EPI is provided by a single-issue in-order core. This is also consistent

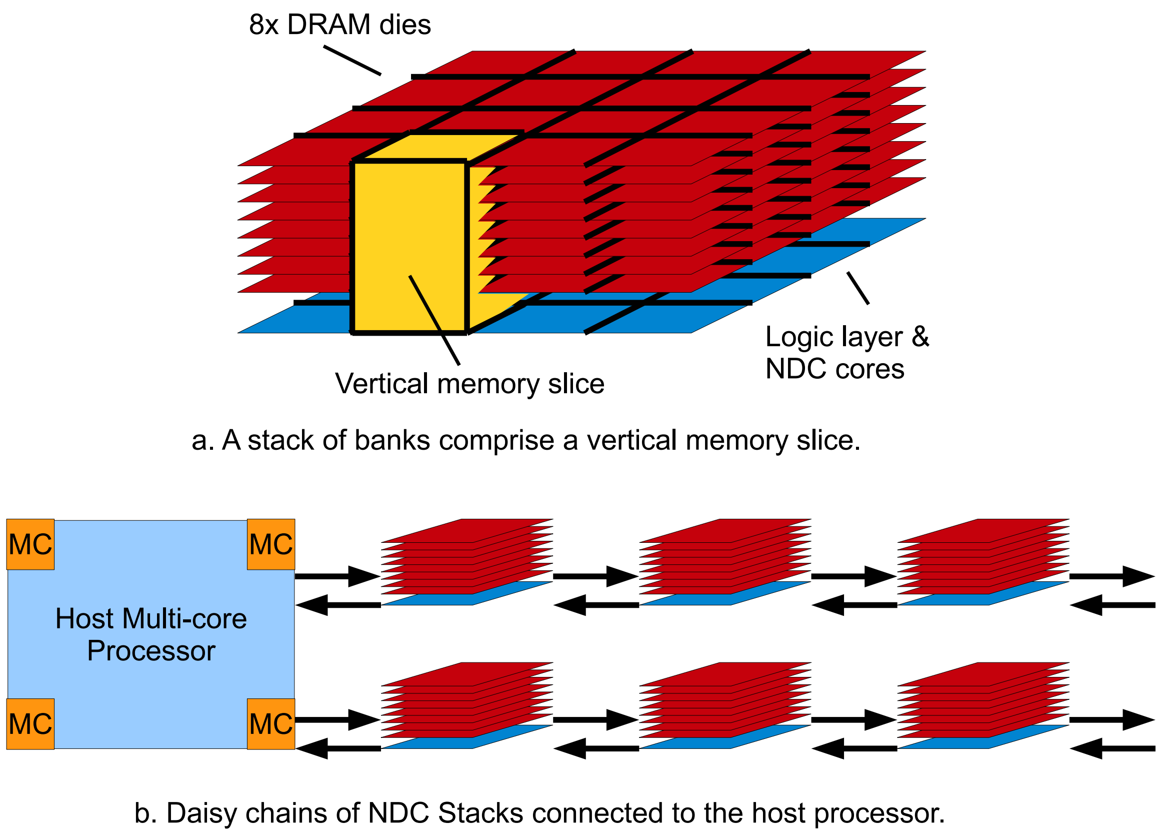

8x DRAM dies
Vertical memory slice
Logic layer & NDC cores
a. A stack of banks comprise a vertical memory slice.
MC
MC
Host Multi-core Processor
MC
MC
b. Daisy chains of NDC Stacks connected to the host processor.

Figure 3.1. The Near Data Computing architecture.

with data on ARM processor specification sheets. We therefore assume an in-order core similar to the ARM Cortex A5 [53].

Parameters for a Cortex A5-like core, and a CMP built out of many such cores, can be found in Table 3.2. Considering that large server chips can be over 400 mm^2 in size, we assume that 512 such cores are accommodated on a server chip (leaving enough room for interconnect, memory controllers, etc.). To construct this table, we calculated the power consumed by on-chip wires to support its off-chip bandwidth, not including the overheads for the intermediate routers [54], we calculated the total power consumed by the on-chip network [55], and factored in the power used by the last level caches and memory controllers [56]. This is a total power rating similar to that of other commercial high-end processors [49].

Table 3.2. Energy Efficient Core (EECore) and baseline systems.

Energy Efficient / ND Core	
Process	32 nm
Power	80 mW
Frequency	1 GHz
Core Type	single-issue in-order
Caches	32 KB I and D
Area (incl. caches)	0.51 mm^2

EE Core Chip Multiprocessor	
Core Count	512
Core Power	41.0 W
NOC Power	36.0 W
LLC and IMC	20.0 W
Total CMP Power	97.0 W

The processor can support a peak total throughput of 512 BIPS and 160 GB/s external read memory bandwidth, i.e., a peak bandwidth of 0.32 read bytes/instruction can be sustained. On such a processor, if the application is compute-bound, then we can build a simpler memory system with DDR3 or DDR4. Our characterization of MapReduce applications shows that the applications are indeed memory-bound. The read bandwidth requirements of our applications range from 0.47 bytes/instruction to 5.71 bytes/instruction. So the HMC-style memory system is required.

We have designed a baseline server that is optimized for in-memory MapReduce workloads. However, this design pays a significant price for data movement: (i) since bandwidth is vital, high-speed SerDes circuits are required at the transmitter and receiver; (ii) since memory capacity is vital to many workloads, daisy chained devices are required, increasing the number of SerDes hops to reach the memory device; (iii) since all the computations are aggregated on large processor chips, large on-chip networks have to be navigated to reach the few high-speed memory channels on the chip.

3.2.2 NDC Hardware

We next show that a more effective approach to handle MapReduce workloads is to move the computation to the 3D-stacked devices themselves. We refer to this as

Near Data Computing to differentiate it from the processing in memory projects that placed logic and DRAM on the same chip and therefore had difficulty with commercial adoption.

While the concept of NDC will be beneficial to any memory bandwidth-bound workload that exhibits locality and high parallelism, we use MapReduce as our evaluation platform in this study. Similar to the baseline, a central host processor with many EECores is connected to many daisy chained memory devices augmented with simple cores. The Map phases of MapReduce workloads exhibit high data locality and can be executed on the memory device; the Reduce phase also exhibits high data locality, but it is still executed on the central host processor chip because it requires random access to data. For random data accesses, average hop count is minimized if the requests originate in a central location, i.e., at the host processor. NDC improves performance by reducing memory latency and by overcoming the bandwidth wall. We further show that the proposed design can reduce power by disabling expensive SerDes circuits on the memory device and by powering down the cores that are inactive in each phase. Additionally, the NDC architecture scales more elegantly as more cores and memory are added, favorably impacting cost.

3.2.2.1 3D NDC Package

As with an HMC package, we assume that the NDC package contains 8 4 Gb DRAM dies stacked on top of a single logic layer. The logic layer has all the interface circuitry required to communicate with other devices, as in the HMC. In addition, we introduce 16 simple processor cores (Near-Data Cores, or NDCores).

3.2.2.2 3D Vertical Memory Slice

In an HMC design, 32 banks are used per DRAM die, with each bank having 16 MB of capacity (assuming a 4 Gb DRAM chip). When 8 DRAM die are stacked on top of each other, 16 banks align vertically to comprise one 3D vertical memory slice, with capacity 256 MB, as seen in Figure 3.1a. Note that a vertical memory slice (referred to as a “vault” in HMC literature) has 2 banks per die. Each 3D vertical memory slice is connected to an NDCore below on the logic layer by Through-Silicon Vias (TSVs). Each NDCore operates exclusively on 256 MB of data, stored in 16 banks directly above it. NDCores have low latency, high bandwidth access to their 3D slice of memory. In the first-generation HMC, there are 1866 TSVs, of which, 512 are used for data transfers at 2 Gb/s each [26].

3.2.2.3 NDCores

Based on our analysis earlier, we continue to use low-EPI cores to execute the embarrassingly parallel Map phase. We again assume an in-order core similar to the ARM Cortex A5 [53]. Each core runs at a frequency of 1 GHz and consumes 80 mW, including instruction and data caches. We are thus adding only 1.28 W total power to the package (and will shortly offset this with other optimizations). Given the spatial locality in the Map phase, we assume a prefetch mechanism that fetches five consecutive cache lines on a cache miss. We also apply this prefetching optimization to all baseline systems tested, not just NDC, and it helps the baseline systems more than the NDC system, due to their higher latency memory access time.

3.2.2.4 Host CPUs and 3D NDC Packages

Because the host processor socket has random access to the entire memory space, we substitute the Shuffle phase with a Reduce phase that introduces a new level of indirection for data access. When the Reduce phase touches an object, it is fetched from the appropriate NDC device (the device where the object was produced by a Mapper). This is a departure from the typical Map, Shuffle, and Reduce pattern of MapReduce workloads, but minimizes data movement when executing on a central host CPU. The Reduce tasks are therefore executed on the host processor socket and its 512 EECores, with many random data fetches from all NDC devices. NDC and both baselines follow this model for executing the Reduce phase.

Having full-fledged traditional processor sockets on the board allows the system to default to the baseline system in case the application is not helped by NDC. The NDCores can remain simple as they are never expected to handle OS functionality or address data beyond their vault. The overall system architecture therefore resembles the optimized HMC baseline we constructed in Section 3.2.1. Each board has 2 CPU sockets. Each CPU socket has 512 low-EPI cores (EECores). Each socket has 8 high-speed links that connect to 4 NDC daisy chains. Thus, every host CPU core has efficient (and conventional) access to the board's entire memory space, as required by the Reduce function.

3.2.2.5 Power Optimizations

Given the two distinct phases of MapReduce workloads, the cores running the Map and Reduce phases will never be active at the same time. If we assume that the cores can be power-gated during their inactive phases, the overall power consumption can be kept in check.

Further, we maintain power-neutrality within the NDC package. This ensures that we are not aggravating thermal constraints in the 3D package. In the HMC package, about 5.7 W can be attributed to the SerDes circuits used for external communication. HMC devices are expected to integrate 4-8 external links and we have argued before that all of these links are required in an optimal baseline. However, in an NDC architecture, external bandwidth is not as vital, because it is only required in the relatively short Reduce phase. To save power, we therefore permanently disable 2 of the 4 links on the HMC package. This 2.85 W reduction in SerDes power offsets the 1.28 W power increase from the 16 NDCores.

The cores incur a small area overhead. Each core occupies 0.51 mm^2 in 32 nm technology. So the 16 cores only incur a 7.6% area overhead, which could also be offset if some HMC links were outright removed rather than just being disabled.

Regardless of whether power-gating is employed, we expect that the overall system will consume less energy per workload task. This is because the energy for data movement has been greatly reduced. The new design consumes lower power than the baseline by disabling half the SerDes circuits. Faster execution times will also reduce the energy for constant components (clock distribution, leakage, etc.).

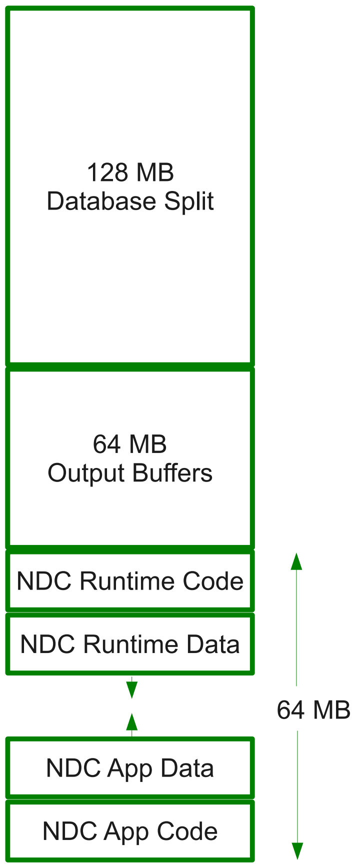
3.2.3 NDC Software

3.2.3.1 User Programmability

Programming for NDC is similar to the programming process for MapReduce on commodity clusters. The user supplies Map and Reduce functions. Behind the scenes, the MapReduce runtime coordinates and spawns the appropriate tasks.

3.2.3.2 Data Layout

Each 3D vertical memory slice has 256 MB total capacity, and each NDCore has access to one slice of data. For our workloads, we populate an NDCore's 256 MB of space with a single 128 MB database split, 64 MB of output buffer space, and 64 MB reserved for code and stack space, as demanded by the application and runtime. Each of these three regions is treated as large superpages. The first two superpages can be accessed by their NDCore and by the central host processor. The third superpage can only be accessed by the NDCore. The logical data layout for one database split is shown in Figure 3.2c.



c. Data layout in an NDC memory slice

Figure 3.2. Data layout in Near Data Computing.

3.2.3.3 MapReduce Runtime

Runtime software is required to orchestrate the actions of the Mappers and Reducers. Portions of the MapReduce runtime execute on the host CPU cores and portions execute on the NDCores, providing functionalities very similar to what might be provided by Hadoop. The MapReduce runtime can serve as a lighweight OS for an NDCore, ensuring that code and data do not exceed their space allocations, and possibly restarting a Mapper on a host CPU core if there is an unserviceable exception or overflow.

3.3 Evaluation

3.3.1 Evaluated Systems

In this work, we compare an NDC-based system to two baseline systems. The first system uses a traditional out-of-order (OoO) multicore CPU, and the other uses a large number of energy-efficient cores (EECores). Both of these processor types are used in a 2-socket system connected to 256 GB of HMC memory capacity, which fits 1024 128 MB database splits. All evaluated systems are summarized in Table 3.3.

Table 3.3. Parameters for evaluated Out-of-Order, EECore, and NDC systems.

Out-of-Order System	
CPU configuration	2x 8 cores, 3.3 GHz
Core parameters	4-wide out-of-order
	128-entry ROB
L1 Caches	32 KB I and D, 4 cycle
L2 Cache	256 KB, 10 cycle
L3 Cache	2 MB, 20 cycle
NDC Cores	—

EECore System	
CPU configuration	2x 512 cores, 1 GHz
Core parameters	single-issue in-order
L1 Caches	32 KB I and D, 1 cycle
NDC Cores	—

NDC System	
CPU configuration	2x 512 cores, 1 GHz
Core parameters	single-issue in-order
L1 Caches	32 KB I and D, 1 cycle
NDC Cores	1024

3.3.1.1 OoO System

On this system, both the Map and Reduce phases of MapReduce run on the high performance CPU cores on the two host sockets. Each of the 16 OoO cores must sequentially process 64 of the 1024 input splits to complete the Map phase. As a baseline, we assume perfect performance scaling for more cores, and ignore any contention for shared resources, other than memory bandwidth, to paint this system configuration in the best light possible.

3.3.1.2 EECore System

Each of the 1024 EECores must compute only one each of the 1024 input splits in a MapReduce workload. Although the frequency of each EECore is much lower than an OoO core, and the IPC of each EECore is lower than an OoO core, the EECore system still has the advantage of massive parallelism, and we show in our results that this is a net win for the EECore system by a large margin.

3.3.1.3 NDCore System

We assume the same type and power/frequency cores for NDCores as EECores. The only difference in their performance is the way they connect to memory. EECores must share a link to the system of connected HMCs, but each NDCore has a direct link to its dedicated memory, with very high bandwidth, and lower latency. This means NDCores will have higher performance than EECores.

In order to remain power neutral compared to the EECore system, each HMC device in the NDC system has half of its 4 data links disabled, and therefore can deliver only half the bandwidth to the host CPU, negatively impacting Reduce performance.

3.3.2 Workloads

We evaluate the Map and Reduce phases of five different MapReduce workloads, namely Group-By Aggregation (*GroupBy*), Range Aggregation (*RangeAgg*), Equi-Join Aggregation (*EquiJoin*), Word Count Frequency (*WordCount*), and Sequence Count Frequency (*SequenceCount*). *GroupBy* and *EquiJoin* both involve a sort, a combine, and a partition in their Map phase, in addition to the Map scan, but the *RangeAgg* workload is simply a high-bandwidth Map scan through the 64 MB database split. These first three workloads use 50 GB of the 1998 World Cup website log [57]. *WordCount* and *SequenceCount* each find the frequency of words or sequences of words in large HTML files, and as input we use 50 GB of Wikipedia HTML data [58]. These last two workloads are more computationally intensive than the others because they involve text parsing and not just integer compares when sorting data.

3.3.3 Methodology

We use a multistage CPU and memory simulation infrastructure to simulate both CPU and DRAM systems in detail.

To simulate the CPU cores (OoO, EE, and NDC), we use the Simics full system simulator [59]. To simulate the DRAM, we use the USIMM DRAM simulator [60], which has been modified to model an HMC architecture. We assume that the DRAM core latency (Activate + Precharge + ColumnRead) is 40 ns. Our simulations model a single Map or Reduce thread at a time and we assume that throughput scales linearly as more cores are used. While NDCores have direct access to DRAM banks, EECores must navigate the memory controller and SerDes links on their way to the HMC device. Since these links are shared by 512 cores, it is important to correctly model contention at the memory controller. A 512-core Simics simulation is not tractable, so we use a trace-based version of the USIMM simulator. This stand-alone trace-based simulation models contention when the memory system is fed memory requests from 512 Mappers or 512 Reducers. These contention estimates are then fed into the detailed single-thread SIMICS simulation.

We wrote the code for the Mappers and Reducers of our five workloads in C, and then compiled them using GCC version 3.4.2 for the simulated architecture. The instruction mix of these workloads is strictly integer-based. For each workload, we have also added 1 ms execution time overheads for beginning a new Map phase, transitioning between Map and Reduce phases, and for completing a job after the Reduce phase. This conservatively models the MapReduce runtime overheads and the cost of cache flushes between phases.

We evaluate the power and energy consumed by our systems taking into account workload execution times, memory bandwidth, and processor core activity rates. We calculate power for the memory system as being equal to the the sum of the power used by each logic layer in each HMC, including SerDes links, the DRAM array background power, and power used to access the DRAM arrays for reads and writes. We assume that the four SerDes links consume a total of 5.78 W per HMC, and the remainder of the logic layer consumes 2.89 W [51]. Total maximum DRAM array power per HMC is assumed to be 4.7 W for 8 DRAM die [26]. We approximate background DRAM array power at 10% of this maximum value [61], or 0.47 W, and the remaining DRAM power is dependent on DRAM activity. Energy is consumed in the arrays on each access at the rate of an additional 3.7 pJ/bit (note that the HMC implements narrow rows and a close page policy [26]). For data that are moved to the processor socket, we add 4.7 pJ/bit to navigate the global wires between the memory controller and the core [54]. This is a conservative estimate because it ignores intermediate routing elements, and favors the EECore baseline. For the core power estimates, we assume that 25% of the 80 mW core peak power can be attributed to leakage (20 mW). The dynamic power for the core varies linearly between 30 mW and 60 mW, based on IPC (since many circuits are switching even during stall cycles).

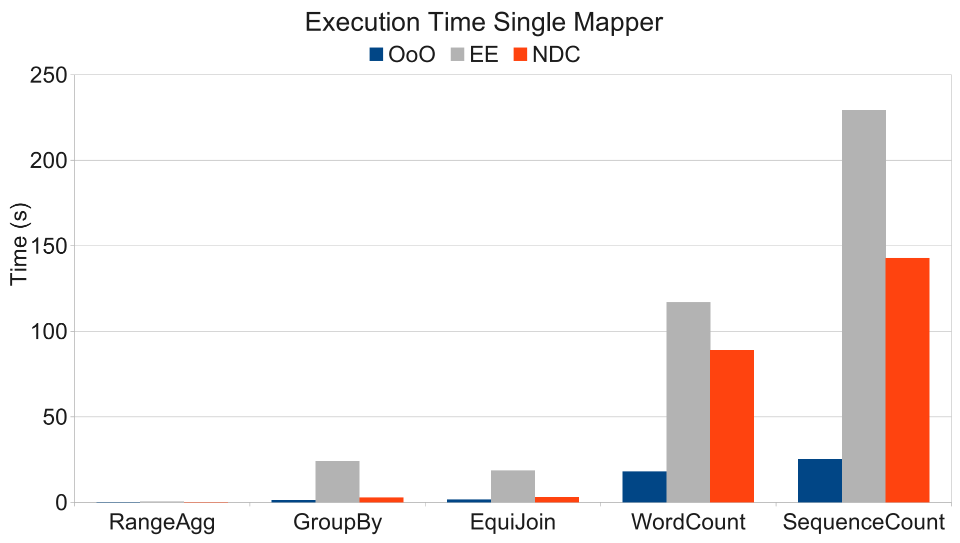
3.4 Performance Results

3.4.1 Individual Mapper Performance

We first examine the performance of a single thread working on a single input split in each architecture. Figure 3.3 shows the execution latency of a single mapper for each workload.

We show both normalized and absolute execution times to show the scale of each of these workloads. When executing on an EECore, a *RangeAgg* Mapper task takes on the order of milliseconds to complete, *GroupBy* and *EquiJoin* take on the order of seconds to complete, and *WordCount* and *SequenceCount* take on the order of minutes to complete.

RangeAgg, *GroupBy*, and *EquiJoin* have lower compute requirements than *Word-Count* and *SequenceCount*, so in these workloads, because of its memory latency advantage, an NDCore is able to nearly match the performance of an OoO core. The EECore system falls behind in executing a single Mapper task compared to both OoO



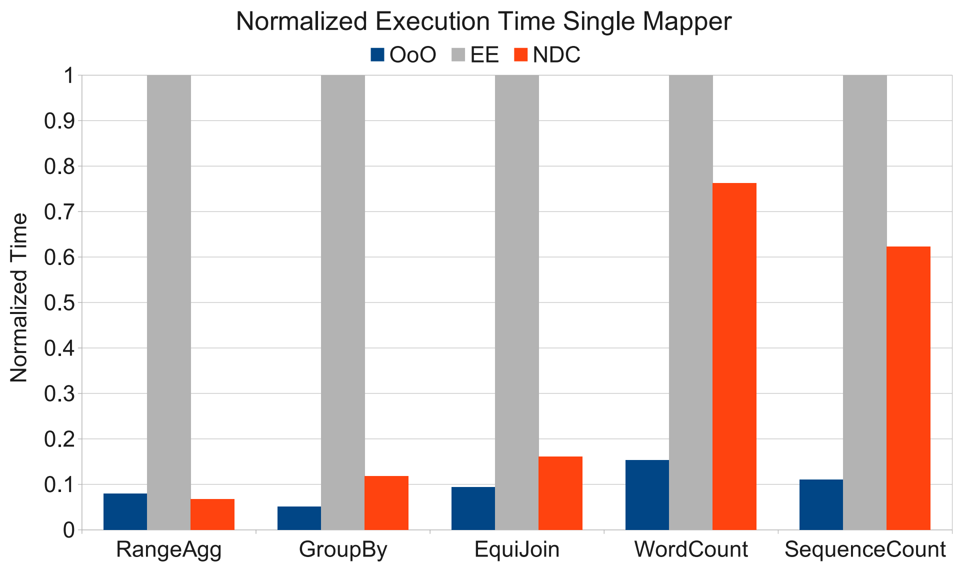


Figure 3.3. Execution times of a single Mapper task, measured in absolute time (top), and normalized to EE execution time (bottom).

and NDCores, because its HMC link bandwidth is maxed out for some workloads, as seen in Section 3.4.3.

3.4.2 Map Phase Performance

Map phase execution continues until all Mapper tasks have been completed. In the case of the EE and NDC systems, the number of Mapper tasks and processor cores is equal, so all Mapper tasks are executed in parallel, and the duration of the Map phase is equal to the time it takes to execute one Mapper task. In the case of the OoO system, Mapper tasks outnumber processor cores 64-to-1, so each OoO processor must sequentially execute 64 Mapper tasks.

Because of this, the single-threaded performance advantage of the OoO cores becomes irrelevant, and both EE and NDC systems are able to outperform the OoO system by a wide margin. As seen in Figure 3.4, compared to the OoO system, the EE system reduces Map phase execution times from 69.4% (*RangeAgg*), up to 89.8%

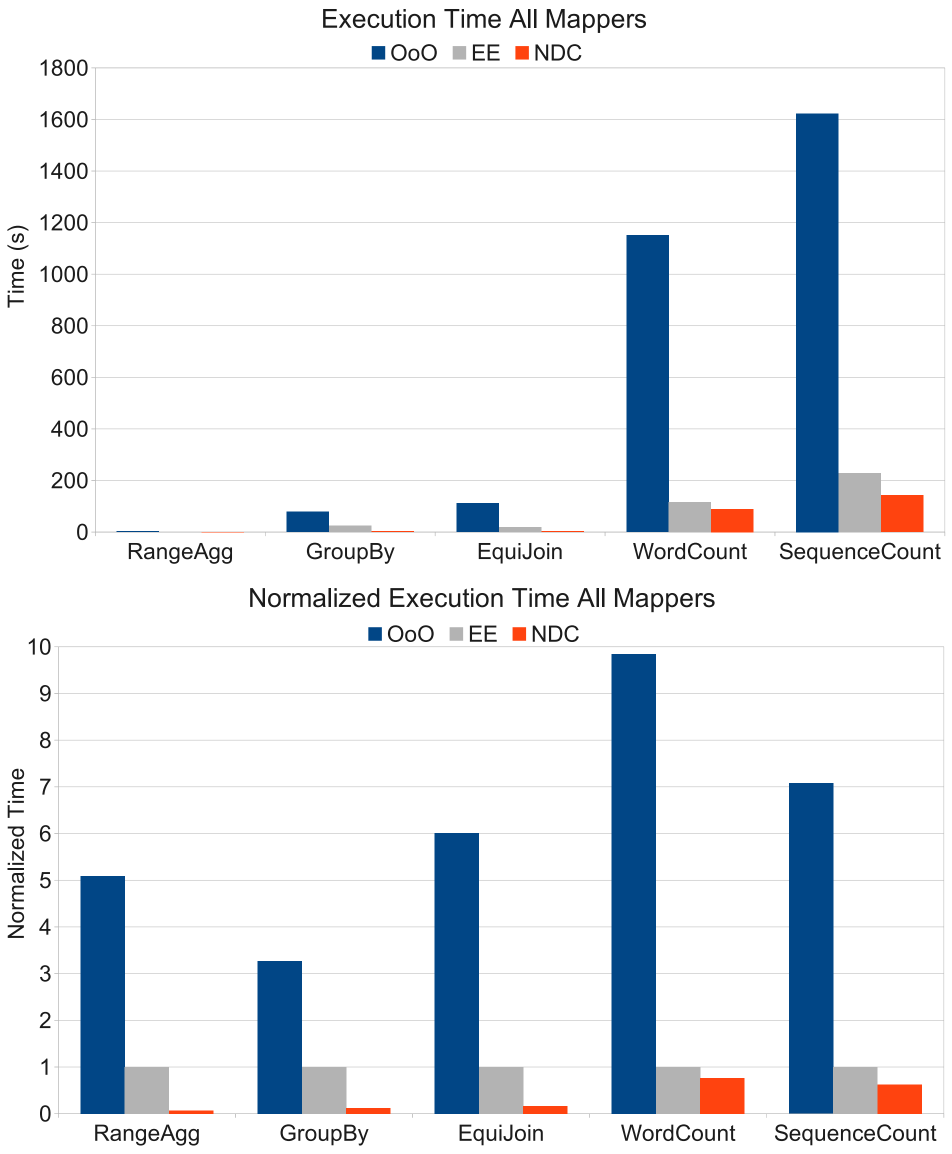


Figure 3.4. Execution times of all Mapper tasks, measured in absolute time (top), and normalized to EE execution time (bottom).

(*WordCount*). The NDC system improves upon the EE system by further reducing execution times from 23.7% (*WordCount*), up to 93.2% (*RangeAgg*).

3.4.3 Bandwidth

The NDC system is able to improve upon the performance of the OoO and EE systems because it is not constrained by HMC link bandwidth during the Map phase. Figure 3.5 shows the read and write bandwidth for each 2-socket system, as well as a

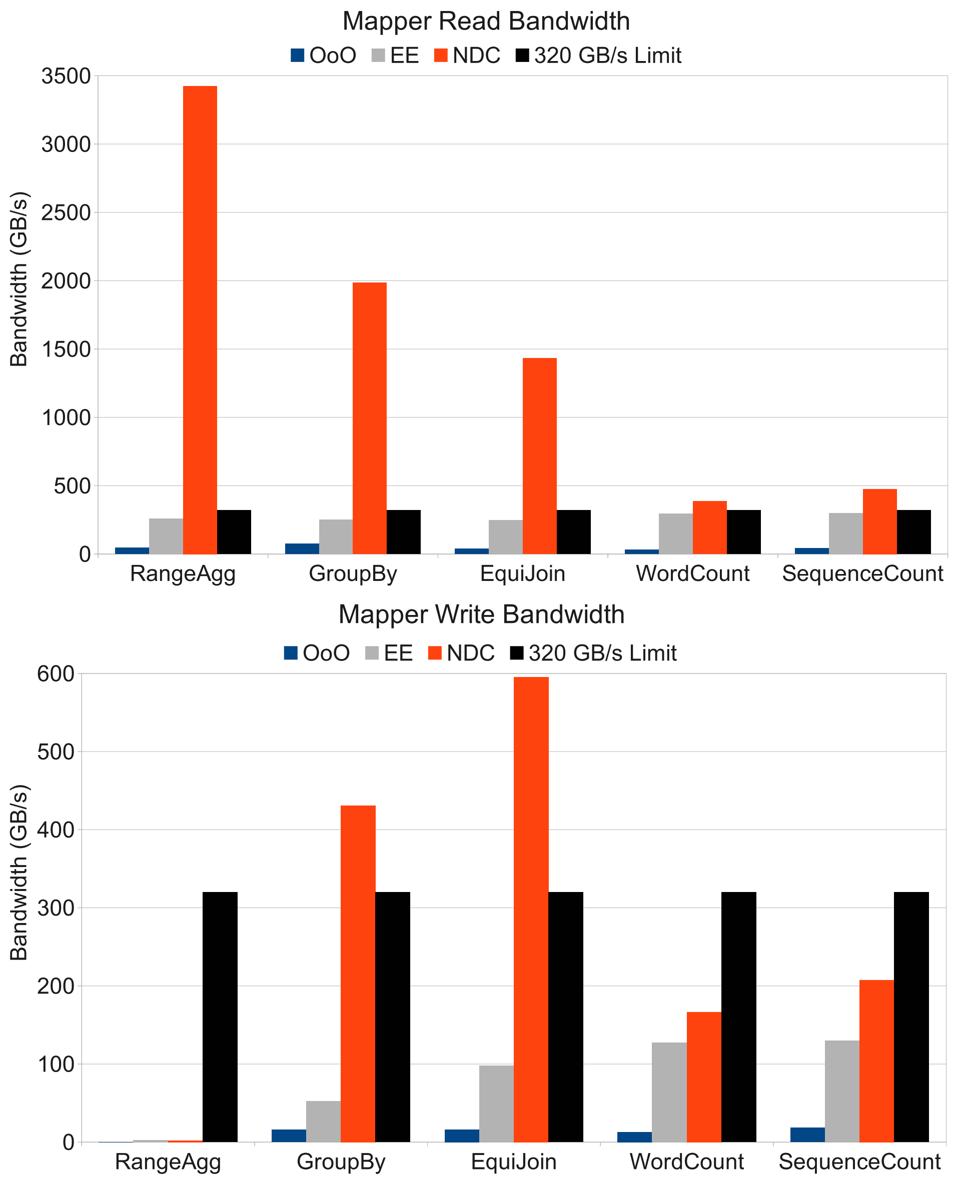


Figure 3.5. Bandwidth usage during Map phase for an entire 2-socket system. Maximum HMC link read and write bandwidth are each 320 GB/s for the system.

bar representing the maximum HMC link bandwidth, which sets an upper bound for the performance of the OoO and EE systems.

The OoO system is unable to ever come close to saturating the available bandwidth of an HMC-based memory system. The EE system is able to effectively use the large amounts of available bandwidth, but because the bandwidth is a limited resource, it puts a cap on the performance potential of the EE system. The NDC system is not constrained by HMC link bandwidth, and is able to use an effective bandwidth many times that of the other systems. While the two baseline systems are limited to a maximum read bandwidth of 320 GB/s, the NDC system has a maximum aggregate TSV bandwidth of 8 TB/s. In fact, this is the key attribute of the NDC architecture – as more memory devices are added to the daisy chain, the bandwidth usable by NDCores increases. On the other hand, as more memory devices are added to the EE baseline, memory bandwidth into the two processor sockets is unchanged.

3.4.4 MapReduce Performance

So far we have focused on the Map phase of MapReduce workloads, because this phase typically dominates execution time, and represents the best opportunity for improving the overall execution time of MapReduce workloads. Figure 3.6 shows how execution time is split between Map and Reduce phases for each workload, and shows the relative execution times for each system.

The OoO and EE systems use the same processing cores for both Map and Reduce phases, but the NDC system uses NDCores for executing the Map phase, and EECores for executing the Reduce phase. Performance improves for both Map and Reduce phases when moving from the OoO system to the EE system, but only Map phase performance improves when moving from the EE system to the NDC system. Reduce phase performance degrades slightly for the NDC system since half the SerDes links are disabled (to save power).

Overall, compared to the OoO system, the EE system is able to reduce MapReduce execution time from 69.4% (*GroupBy*), up to 89.8% (*WordCount*). NDC further reduces MapReduce execution times compared to the EE system from 12.3% (*Word-Count*), up to 93.2% (*RangeAgg*).

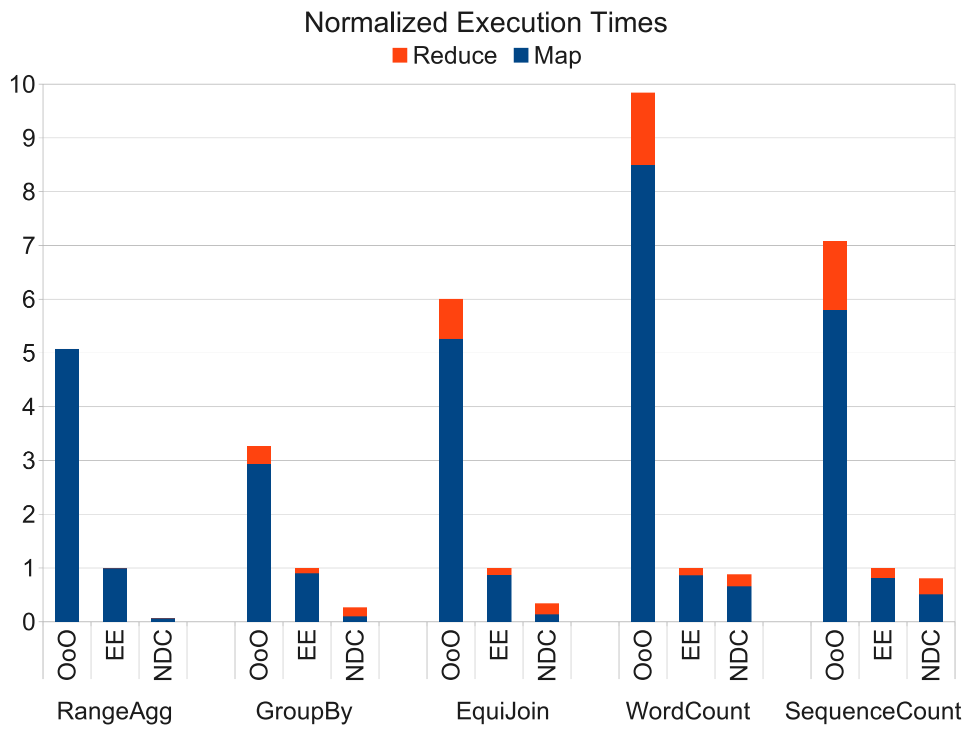


Figure 3.6. Execution time for an entire MapReduce job normalized to the EE system.

3.4.5 Energy Consumption

We consider both static and dynamic energy in evaluating the energy and power consumption of EE and NDC systems. Figure 3.7 and Figure 3.8 show the breakdown in energy consumed by the memory subsystem and the processing cores. Figure 3.7 shows the energy savings when moving from an EE system to an NDC system that uses a full complement of HMC links (NDC FL). Compared to the EE system, the NDC FL system reduces energy consumed to complete an entire MapReduce task from 28.2% (*WordCount*), up to 92.9% (*RangeAgg*). The processor and memory energy savings primarily come from completing the tasks more quickly.

Figure 3.8 assumes NDC FL as a baseline and shows the effect of various power optimizations. NDC Half Links is the NDC system configuration we use in all of our other performance evaluations, and is able to reduce energy consumed by up to 23.1% (*RangeAgg*) compared to NDC Full Links. Disabling half the links reduces performance by up to 22.6% because it only affects the Reduce phase (as seen in the dark bars in Figure 3.6). NDC-PD is a model that uses all the SerDes links, but

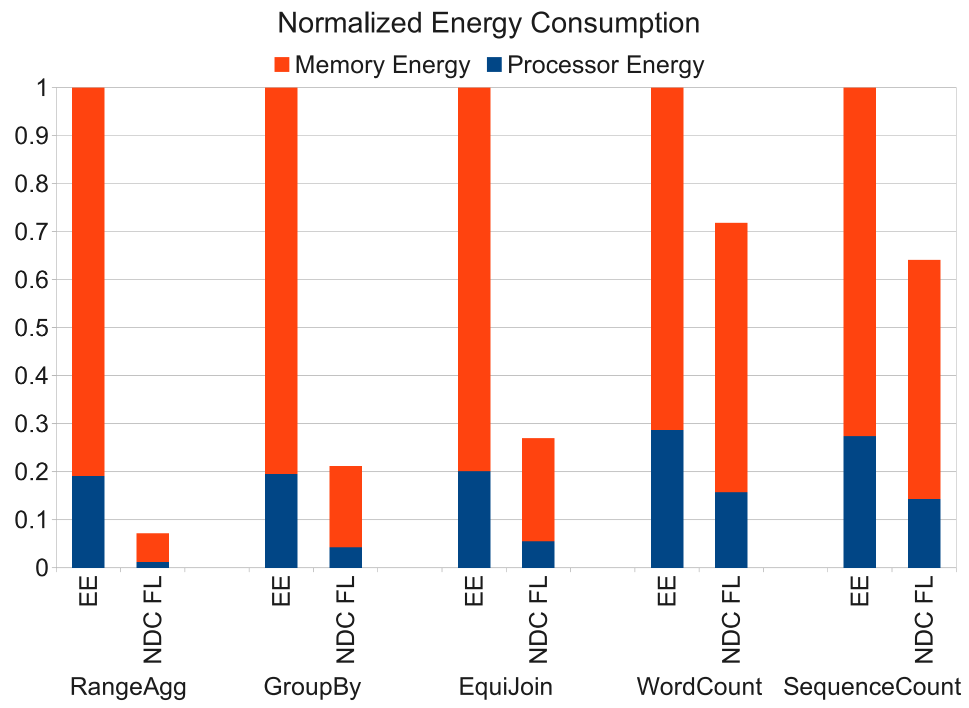


Figure 3.7. Energy consumed by the memory and processing resources, normalized to EE processor energy.

places unutilized cores in power-down modes. So NDCores are powered down during the Reduce phase and EECores are powered down during the Map phase. We assume that a transition to low-power state incurs a 1.0 ms latency and results in core power that is 10% of the core peak power. Note that the transition time is incurred only once for each workload and is a very small fraction of the workload execution time, which ranges between dozens of milliseconds to several minutes. This technique is able to reduce overall system energy by up to 10.0% (*SequenceCount*). Finally, combining the Half Links optimization with core power-down allows for energy savings of 14.7% (*GroupBy*) to 28.3% (*RangeAgg*).

3.4.6 HMC Power Consumption and Thermal Analysis

In addition to a system-level evaluation of energy consumption, we also consider the power consumption of an individual HMC device. In the EE system, the HMC device is comprised of a logic layer, including 4 SerDes links, and 8 vertically stacked DRAM dies. An NDC HMC also has a logic layer and 8 DRAM dies, but it only uses 2 SerDes links and also includes 16 NDC cores. As with the energy consumption

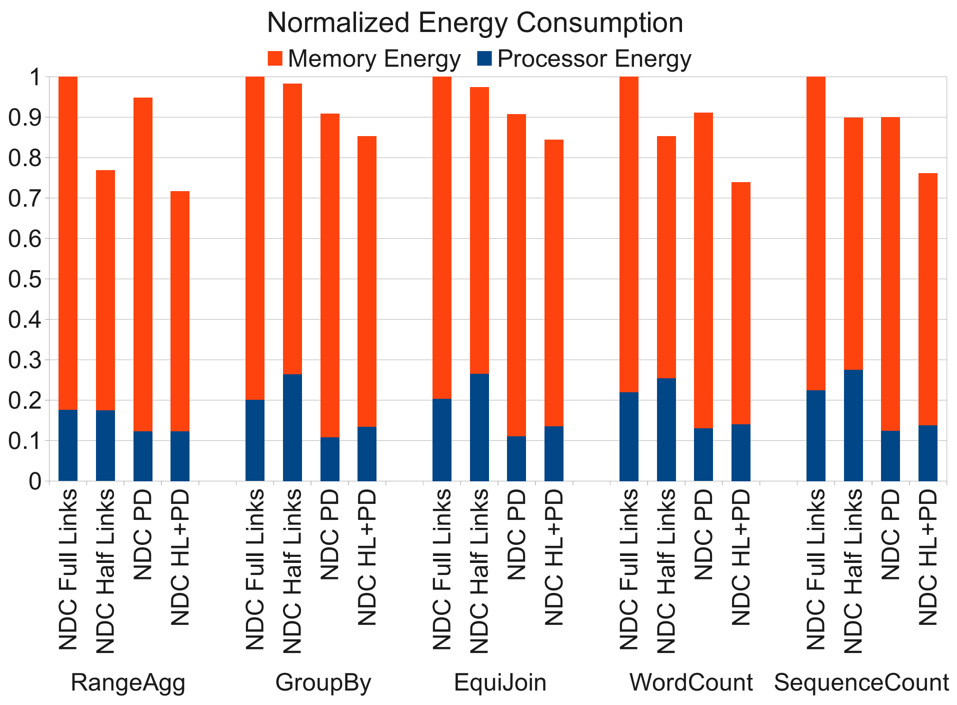


Figure 3.8. Energy consumed by the memory and processing resources, normalized to NDC FL processor energy.

evaluation, we consider core and DRAM activity levels in determining HMC device power. Figure 3.9 shows the contribution of HMC power from the logic layer, the DRAM arrays, and NDC cores, if present.

The baseline HMCs do not have any NDC cores, so they see no power contribution from that source, but they do have twice the number of SerDes links, which are the single largest consumer of power in the HMC device.

The NDC design saves some power by trading 2 SerDes links for 16 NDCores. However, we also see an increase in DRAM array power in NDC. In the EECore baseline, host processor pin bandwidth is shared between all HMCs in the chain, and no one HMC device is able to realize its full bandwidth potential. This leads to a low power contribution coming from DRAM array activity, because each HMC device can contribute on average only 1/8th the bandwidth supported by the SerDes links. The NDC architecture, on the other hand, is able to keep the DRAM arrays busier by utilizing the available TSV bandwidth. Overall, the NDC HMC device consumes up to 16.7% lower power than the baseline HMC device.

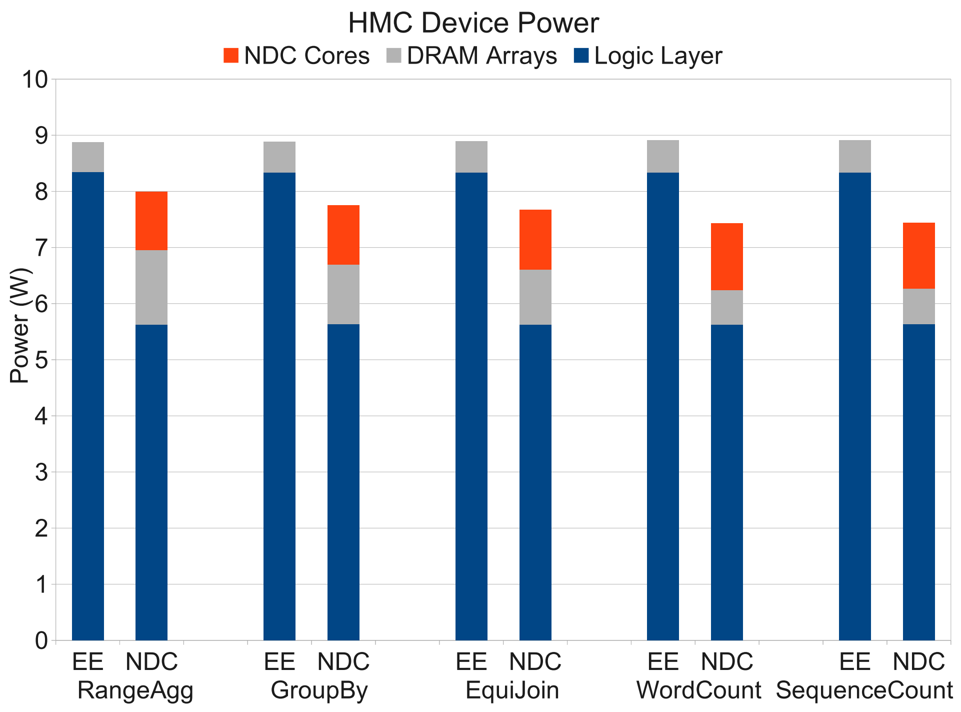


Figure 3.9. Breakdown of power consumed inside an HMC stack for both EE and NDC systems. The HMC in the EE system contains no NDC cores, and the HMC in the NDC system uses half the number of data links.

We also evaluated the baseline HMC and NDC floorplans with Hotspot 5.0 [62], using default configuration parameters, an ambient temperature of 45C inside the system case, and a heat spreader of thickness 0.25 mm. We assumed that each DRAM layer dissipates 0.59 W, spread uniformly across its area. The logic layer's 8.67 W is distributed across various units based on HMC's power breakdown and floorplan reported by Sandhu [51]. We assumed that all 4 SerDes links were active. For each NDCore, we assumed that 80% of its 80 mW power is dissipated in 20% of its area to model a potential hotspot within the NDCore. Our analysis showed a negligible increase in device peak temperature from adding NDCores. This is shown by the logic layer heatmap in Figure 3.10; the SerDes units have much higher power densities than the NDCore, so they continue to represent the hottest units on the logic chip. We carried out a detailed sensitivity study and observed that the NDCores emerge as hotspots only if they consume over 200 mW each. The DRAM layers exceed 85C (requiring faster refresh) only if the heat spreader is thinner than 0.1 mm.

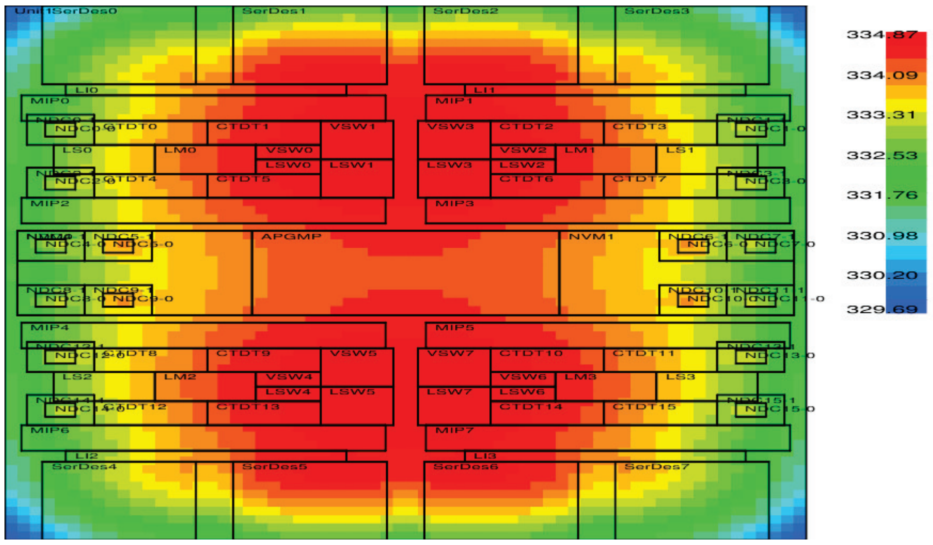


Figure 3.10. Heatmap of the logic layer in the NDC system.

3.5 Chapter Conclusions

This chapter argues that the concept of Near-Data Computing is worth revisiting in light of various technological trends. We argue that the MapReduce framework is a good fit for NDC architectures. We present a high-level description of the NDC hardware and accompanying software architecture, which presents the programmer with a MapReduce-style programming model. We first construct an optimized baseline that uses daisy chains of HMC devices and many energy-efficient cores on a traditional processor socket. This baseline pays a steep price for data movement. The move to NDC reduces the data movement cost and helps overcome the bandwidth wall. This helps reduce overall workload execution time by 12.3% to 93.2%. We also employ power-gating for cores and disable SerDes links in the NDC design. This ensures that the HMC devices consume less power than the baseline and further bring down the energy consumption. Further, we expect that NDC performance, power, energy, and cost will continue to improve as the daisy chains are made deeper.

CHAPTER 4

NEAR DATA COMPUTING USING COMMODITY MEMORY

One of the primary reasons for the resurgence of interest in Near Data Computing (NDC) is the recent emergence of 3D-stacked memory+logic products, such as Micron's Hybrid Memory Cube (HMC) [26]. Such devices enable co-location of processing and memory in a single package, without impacting the manufacturing process for individual DRAM dies and logic dies. The high bandwidth between the logic and memory dies with through-silicon vias (TSVs) can enable significant speedups for memory-bound applications.

This paper first describes the basic NDC framework that appeared in Chapter 3. It then extends that work by considering compelling alternatives. One of these alternative architectures eliminates the use of HMC devices because of their expected high purchase price. The primary contribution of this work is to generalize the NDC approach and to compare the merits of these different implementations against a highly optimized non-NDC baseline.

4.1 Background

4.1.1 MapReduce Workloads

MapReduce applications typically operate on disk-resident data. Large datasets (often key-value pairs) are partitioned into splits. Splits are distributed across several nodes of a commodity cluster. Users provide Map and Reduce functions to aggregate information from these large datasets. A run-time framework, e.g., Hadoop, is responsible for spawning Map and Reduce tasks, moving data among nodes, handling fault tolerance, etc.

In this paper, we consider implementations of the Map and Reduce phases which we have written in C. These codes are run as stand-alone threads in our simulators.

Our simulators are not capable of executing the JVM and Hadoop run-times, so they are simply modeled as constant overheads at the start and end of each Map/Reduce task. We assume that each split is 128 MB in size, and is resident in DRAM memory. We execute workloads that have varying bandwidth requirements, ranging from 0.47 bytes/instruction to 5.71 bytes/instruction.

4.1.2 An Optimized Baseline for MapReduce

We now describe a highly optimized baseline for the embarrassingly parallel and bandwidth-intensive in-memory MapReduce workloads.

The processor socket is designed with a large number of low energy-per-instruction (EPI) cores. This will maximize the number of instructions that are executed per joule, and will also maximize the number of instructions executed per unit time, within a given power budget (assuming an embarrassingly parallel workload). We therefore assume an in-order core with power, area, and performance characteristics similar to the ARM Cortex A5 that consumes 80 mW at 1 GHz. Our analysis shows that 512 80 mW cores can be accommodated on a single high-performance chip while leaving enough area and power budget to accommodate an on-chip network, shared LLC, and memory controllers.

This many-core processor must be equipped with the most bandwidth-capable memory interface to maximize performance. The HMC uses high-speed SerDes links to connect to the processor. Using only 128 pins, the HMC can provide 80 GB/s of bandwidth to the processor socket. The HMC also provides low energy per bit (10.48 pJ/bit vs. DDR3's 70 pJ/bit). An important factor in the HMC's energy efficiency is its ability to fetch a cache line from a single DRAM die in the stack, thus limiting the overfetch that plagues a traditional DDR3 memory interface. We therefore connect our many-core processor to 4 HMC devices with 8 high-speed links (each comprised of 16 data bits in each direction). Given the limited capacity of an HMC device (4 GB), we increase the memory capacity of the board by connecting each HMC to other HMC devices in a daisy chain. We assume 4 daisy chains per processor socket, with 8 HMCs in each daisy chain.

Such a baseline maximizes the processing capability and bandwidth for our workload, while retaining the traditional computation model that separates computation and memory to different chips on a motherboard. We refer to this baseline as the EECore model (energy-efficient cores). Such a baseline expends low amounts of energy in computations (low EPI ARM cores) and DRAM array reads (HMC with low overfetch), but pays a steep price for data movement (multiple high-frequency SerDes hops on the daisy chain and on-chip network navigation on the processor). Such an optimized baseline thus further exposes the communication bottleneck and stands to benefit more from NDC.

4.2 Near Data Computing Architectures

4.2.1 NDC with HMCs

In this section, we briefly review the NDC architecture that was described in the previous chapter.

The NDC architecture is built upon a connected network of HMC devices, each of which has 16 80 mW energy-efficient cores placed on its logic chip. These are referred to as Near Data Cores, or NDCores, and we now refer to this augmented HMC as an NDC device.

Each NDCore is tightly coupled to one of the 16 vaults on an NDC device. The vault is a high-bandwidth connection to DRAM banks located directly above the NDCore. The vault has a capacity of 256 MB, providing enough room to store one data split, intermediate MapReduce outputs, and any code/data belonging to the MapReduce threads and the run-time. The NDCore is connected to an L1 I and D cache. On a cache miss, the data are fetched directly from the DRAM arrays directly above. We employ a simple prefetcher for all evaluated systems that prefetches five total cache lines on a cache miss. In essence, each NDCore and its vault represent a self-contained mininode that is capable of performing a single Map or Reduce task with very high efficiency.

While the earlier chapter uses a similar topology to the baseline (host processor sockets connected to daisy chained NDC devices), in this work, we consider a board that only uses NDC devices and eliminates the host processor socket. This avoids having to deal with the requirement that inactive cores be disabled to save power. It also enables an apples-to-apples comparison with the NDC-Module design that will be discussed in the next subsection.

We consider both mesh and ring networks to connect the many NDC devices. The mesh requires 2x the number of SerDes links as the ring network. In both cases, the network serves only to facilitate the Shuffle phase.

4.2.2 NDC with Modules

Our new approach to NDC replaces an HMC-based memory system with a memory system comprised of commodity LPDDR2 x32 chips which are connected via conventional, non-SerDes memory interfaces to low power multicore processors. These processor cores have the same performance and power characteristics as NDCores. Unlike a traditional server board where the processor socket is on the motherboard and the memory chips are on add-in DIMMs, the lightweight processor and its associated memory are both soldered onto add-in daughter card modules that are in turn arranged into networks on the motherboard, as in Figure 4.1. We label this approach NDC-Module. Map and Reduce phases both again execute on the same processors, with an explicit Shuffle phase between them.

The HMC-based NDC devices are expensive, because of their logic chip and the 3D manufacturing process. LPDDR2 DRAM is cheaper, but compared to NDC-HMC, NDC-Module offers less memory bandwidth to each core, so there is not as great an opportunity for high performance. We design the NDC-Module system to have the same memory capacity and number of cores as the NDC-HMC system.

The key insights behind the NDC-Module system are: (i) A standard DDRx channel exposes a small fraction of DRAM chip bandwidth to the processor, because each memory controller pin is shared by many DRAM chips. Moving compute to the DIMM, as in the NDC-Module design, enables available bandwidth to scale linearly with the number of DRAM chips. (ii) Similar to the HMC, an entire cache line is fetched from a single LPDDR2 chip, reducing activation energy and overfetch.

4.2.2.1 NDC-Module Cards

Each NDC-Module card is comprised of 8 nodes. Each node contains one 8 Gb LPDDR2 x32 chip and one 4-core energy-efficient CMP, similar to the memory interface and core count of an nVidia Tegra 3 mobile processor. In total, each card

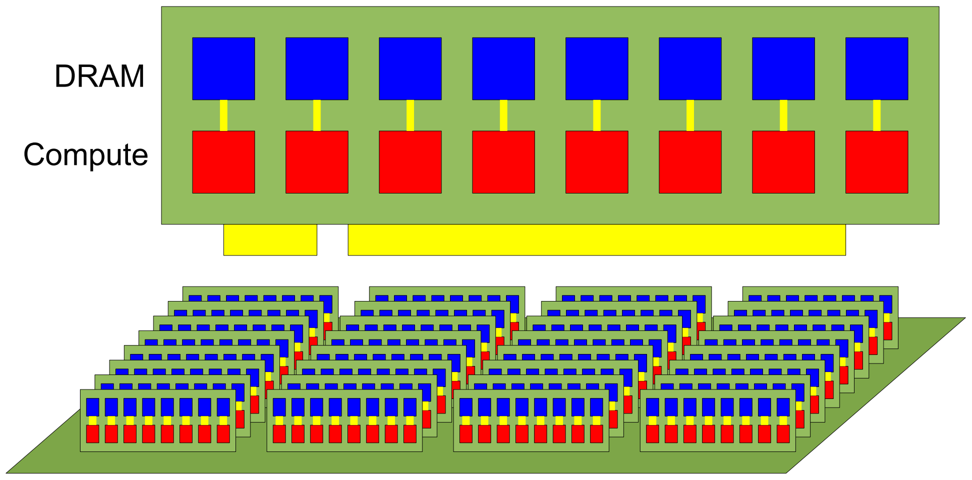


Figure 4.1. The NDC-Module architecture. A single card (top), and a fully populated system board (bottom).

contains 8 GB of LPDDR2 DRAM and 32 energy-efficient cores. This is the same ratio of compute to DRAM as in NDC-HMC.

4.2.2.2 LPDDR2 x32 DRAM Chips

The NDC-Module system uses commodity LPDDR2 DRAM chips, instead of an HMC-based memory system. We use LPDDR2 x32 chips which can transmit 1066 Mb/s per IO wire, for a total bandwidth of 4.26 GB/s per chip. This bandwidth is shared between read and write operations, and services only 4 energy efficient cores. Each node's bandwidth is independent of other nodes. This gives a higher per-core theoretical bandwidth than that of the baseline EECore system, but lower per-core bandwidth than the NDC-HMC system.

4.2.2.3 NDC-Module Motherboard Organization

As seen in Figure 4.1, we organize 8 nodes on a single add-in card for a total of 8 GB and 32 low-EPI cores. The NDC-Module system is made up of 32 such add-in modules arranged on a single motherboard. We employ PCI-Express 3.0 x16 to connect all of the modules, offering a total of 31.5 GB/s of bandwidth with which to perform the Shuffle phase.

4.3 Evaluation

4.3.1 Evaluated Systems

In this work, we compare NDC-HMC, with both ring- and mesh-network configurations, to NDC-Module, and our optimized socket-based baseline (EECore). All systems have equal core counts (1024) and memory capacity (256 GB, storing 1024 128 MB splits), but vary in the connections between processor and memory, and how NDC devices are networked. All evaluated systems are summarized in Table 4.1.

4.3.2 Methodology

We use a multistage CPU and memory simulation infrastructure to simulate both CPU and DRAM systems in detail.

To simulate the CPU cores (EE and NDC), we use the Simics full system simulator. To simulate the DRAM, we use the USIMM DRAM simulator [60], which has been modified to model an HMC architecture. We assume that the DRAM core latency (Activate + Precharge + ColumnRead) is 40 ns.

Table 4.1. Parameters for evaluated EECore, NDC-HMC, and NDC-Module systems.

EECore System	
CPU configuration	2x 512 cores, 1 GHz
Core parameters	single-issue in-order
L1 Caches	32 KB I and D, 1 cycle
NDC Cores	—
NDC-HMC System	
NDC Cores	1024 cores, 1 GHz
Core parameters	single-issue in-order
L1 Caches	32 KB I and D, 1 cycle
Inter-Device Connection	8x8 Mesh or 1D Ring
NDC-Module System	
Add-In Cards (AIC)	32
AIC Configuration	8 NDC-Module Nodes
NDC-Module Node	4 low-EPI cores, 1 GB
Total NDC Cores	1024 cores, 1 GHz
Core parameters	single-issue in-order
L1 Caches	32 KB I and D, 1 cycle
Inter-AIC Connection	PCI-Express 3.0 x16

Our CPU simulations model a single Map or Reduce thread, and we assume that throughput scales linearly as more cores are used. For the EECore system, 512-core Simics simulations are not tractable, so we use a trace-based version of the USIMM simulator to model memory contention, and then feed these contention estimates into detailed single-thread Simics simulations.

All power and energy evaluations take into account workload execution times and component activity rates.

We assume that the 4 SerDes links of an HMC consume a total of 5.78 W per HMC, and the remainder of the logic layer consumes 2.89 W. Total maximum DRAM array power per HMC is assumed to be 4.7 W for 8 DRAM die [26]. We approximate background DRAM array power at 10% of this maximum value, or 0.47 W according to the Micron power calculator, and the remaining DRAM power is dependent on DRAM activity. Energy is consumed in the arrays on each access at the rate of an additional 3.7 pJ/bit.

For LPDDR2, bit-transport energy is 40 pJ/bit, and when operating at peak bandwidth, one LPDDR2 chip consumes 1.36 W of power. Background power for LPDDR2 chips is assumed to be 42.3 mW.

For data that are moved to the processor socket, we add 4.7 pJ/bit to navigate the global wires between the memory controller and the core. This is a conservative estimate because it ignores intermediate routing elements, and favors the EECore baseline.

For the core power estimates, we assume that 25% of the 80 mW core peak power can be attributed to leakage (20 mW). The dynamic power for the core varies linearly between 30 mW and 60 mW, based on IPC (since many circuits are switching even during stall cycles).

4.3.3 Workloads

We evaluate the Map and Reduce phases of 5 different MapReduce workloads, namely Group-By Aggregation (GroupBy), Range Aggregation (RangeAgg), Equi-Join Aggregation (EquiJoin), Word Count Frequency (WordCount), and Sequence Count Frequency (SequenceCount).

GroupBy and EquiJoin both involve sorting as part of their Map phase, but the RangeAgg workload is simply a high-bandwidth Map scan through the input split. These first three workloads use the 1998 World Cup website log [57] as input.

WordCount and SequenceCount find the frequency of words or sequences of words in large HTML files, and as input we use Wikipedia HTML data [58]. These workloads also involve sorting, but parsing and sorting text items is more computationally intensive than sorting integers, so these workloads are more compute-bound than the others.

When executing on the EECore system, a RangeAgg MapReduce task takes on the order of milliseconds to complete, GroupBy and EquiJoin take on the order of seconds to complete, and WordCount and SequenceCount take on the order of minutes to complete.

For each workload, we have also added 1 ms execution time overheads for beginning a new Map phase, transitioning between Map and Reduce phases, and for completing a job after the Reduce phase. This conservatively models the MapReduce runtime overheads and the cost of cache flushes between phases. Even if these estimates are off by three orders of magnitude, they would still have a negligible impact on the longer running workloads.

4.4 Performance Results

4.4.1 Bandwidth

Unlike the EECore system, The NDC systems are not limited by host CPU processor socket bandwidth during Map phase execution, as seen in Figure 4.2. In RangeAgg, we see that the NDC-HMC system is able to attain an average of 3.4 TB/s of aggregate read bandwidth, the NDC-Module system is able to attain an average of 580 GB/s aggregate read bandwidth, but the EECore system is limited to only using 260 GB/s of aggregate read bandwidth.

We see that RangeAgg is memory bandwidth-bound, that WordCount and SequenceCount are compute-bound, and that GroupBy and EquiJoin fall somewhere in between. All of the performance advantage, and much of the energy advantage, of the NDC systems over the EECore system comes from high-bandwidth access to memory, as we will now show.

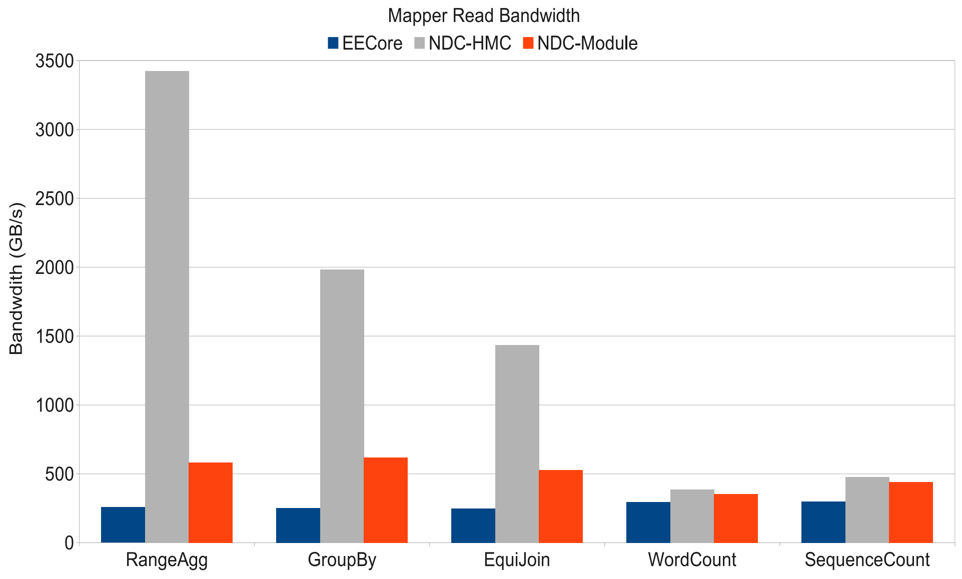


Figure 4.2. Average read bandwidth usage during Map phase for an entire system.

4.4.2 MapReduce Performance

Figure 4.3 shows how execution time is split between Map, Shuffle, and Reduce phases for each workload, and shows the relative execution times for each system. Map phase performance directly tracks with the memory bandwidth numbers seen above for each system and workload, so the NDC-HMC systems have the highest performance, followed by the NDC-Module system.

The Shuffle phase generally takes a very small amount of overall MapReduce execution time. Only when executing the GroupBy and EquiJoin workloads does the Shuffle phase have a nontrivial impact on performance. However, even in the worst case, performance for EquiJoin is hurt by only 8.4% when using NDC-HMC Ring, versus NDC-HMC Mesh. The EECore system does not use the Shuffle phase at all. Instead, the EE system uses long-latency random accesses to memory during the Reduce phase.

Compared to the EECore baseline, NDC-HMC Mesh and NDC-HMC Ring both improve performance by 23.7% (WordCount) to 92.7% (RangeAgg). NDC-Module improves performance over the EECore system by 16.5% (WordCount) to 55.5% (RangeAgg).

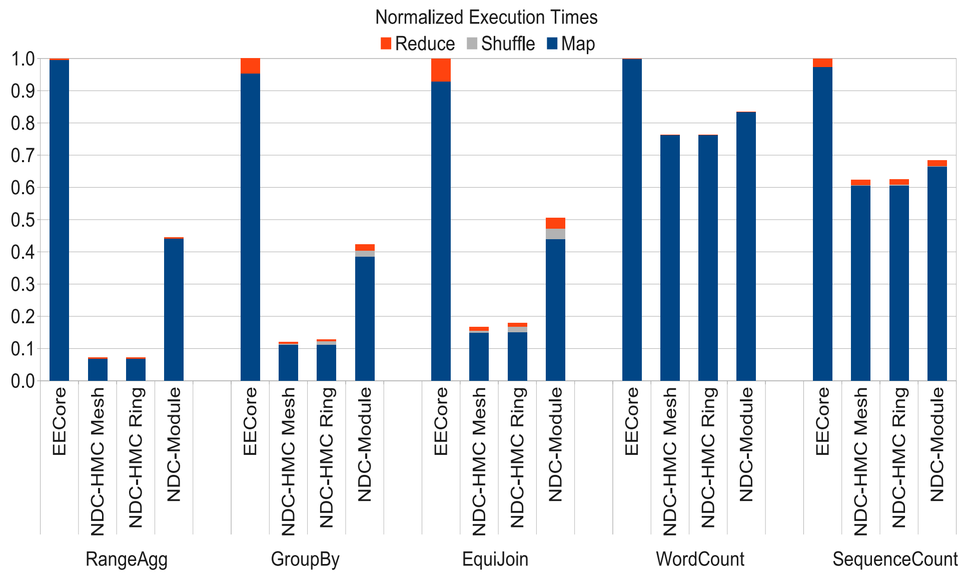


Figure 4.3. Execution time for an entire MapReduce job normalized to the EE system.

4.4.3 Energy Consumption

We consider both static and dynamic energy in evaluating the energy and power consumption of EE and NDC systems. Figure 4.4 shows the breakdown in energy consumed by the memory subsystem, and the processing cores. All three NDC systems improve upon the energy efficiency of the EE baseline, in large part because of improved task execution times.

NDC-HMC Ring improves upon the energy efficiency of NDC-HMC Mesh because it uses fewer SerDes links. The gap between the Mesh and Ring systems is narrowest for GroupBy and EquiJoin, which spend longer in their Shuffle phase, the only phase where having more SerDes links helps performance.

The NDC-Module system uses LPDDR2 which has a higher energy cost per access, compared to an HMC-based memory, but lower background power, due to its lack of a SerDes interface. The compute-bound WordCount and SequenceCount workloads have many opportunities for the LPDDR2 interface to be idle, so the NDC-Module system uses the least energy in these two workloads.

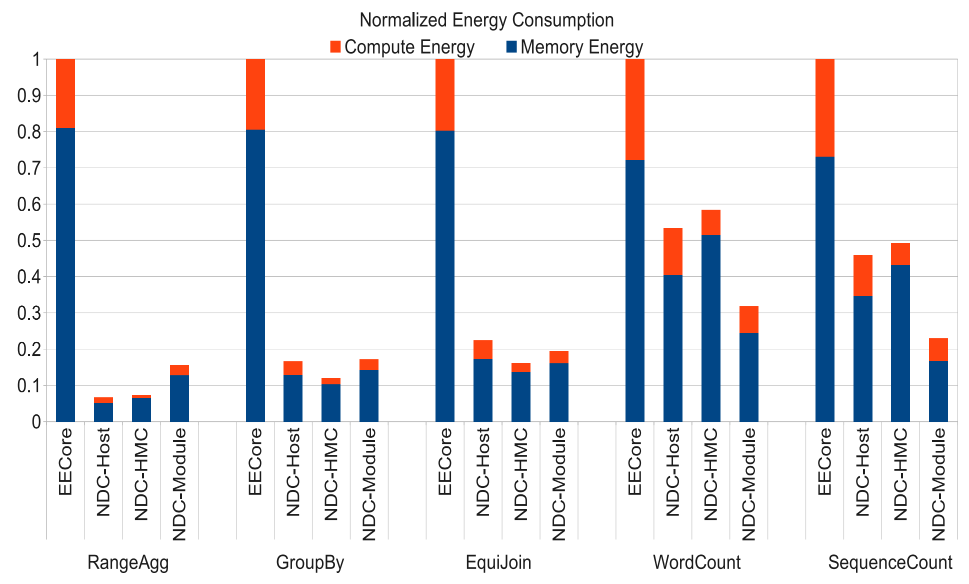


Figure 4.4. Energy consumed by the memory and processing resources, normalized to the EE system.

4.4.4 Cost

We now compare the purchase cost of the EE and NDC-Module systems. We conservatively assume that since the NDC-Module system uses 8x the total memory pins in the system, that it also uses 8x the total silicon area for CPUs. We assume the EE system uses 2x 435 mm^2 CPUs, and that the NDC-Module system uses 256x 27 mm^2 CPUs. Further, assuming \$5000 for a processed 8 inch wafer, and 80% yield rates, the manufacturing cost of each EE CPU would be \$135, for a total system CPU cost of \$270, and the cost of each NDC-Module CPU would be \$5.80, for a total system CPU cost of \$1485.

If LPDDR2 x32 memory is available at a cost of \$10/GB, then the NDC-Module system would need to spend \$2560 on DRAM. Under these assumptions, the break-even point at which both EE and NDC-Module systems would spend the same on CPU and memory devices is if HMCs are only 1.47x the cost of LPDDR2 for the same capacity. However, HMCs have the additional costs of 3D stacking and a logic layer, and if HMCs are more expensive than 1.47x the cost of LPDDR2 for the same capacity, then the NDC-Module system will spend less on CPU and memory devices, in addition to the benefits of increased performance and lower energy use.

If the purhcase price for HMCs is high enough, it could make it infeasible to build an entire main memory out of HMC devices, and render our EECore baseline completely impractical. In such a scenario, the NDC-HMC system may still be attractive despite the high price of HMCS, because of its greatly increased performance and reduced energy consumption. However, if that also proves too expensive, then the NDC-Module system, using only commodity DRAM, would be the most attractive option.

4.5 Chapter Conclusions

This paper investigates implementing NDC without a host CPU. NDC-HMC is built out of a network of HMC-based NDC devices, and NDC-Module is built out of add-in cards that combine low-power multiprocessors with LPDDR2 x32 memory. Both system architectures improve performance in MapReduce applications by reducing the contention for shared memory bandwidth. NDC-Module has lower performance than NDC-HMC, but given its use of commodity DRAM, and lower energy use in compute-bound workloads, it is still an attractive design point.

All proposed NDC implementations offer large performance benefits and energy savings when compared to systems built with processor sockets and shared memory channels. Compared to the EECore baseline, NDC-HMC reduces MapReduce task execution times by up to 92.7%, and uses up to 94.1% less energy, and NDC-Module reduces MapReduce task execution times by up to 55.5%, and uses up to 84.4% less energy.

CHAPTER 5

NEAR DATA COMPUTING AND FIXED FUNCTION ACCELERATORS

As a throughput architecture for MapReduce workloads, the NDC Module system achieves high performance and low energy consumption without using an exotic main memory solution, but it still must pay the overheads of a generally programmable computer (e.g., fetch and decode of RISC-style instructions). MapReduce requires general programmability, because a user (programmer) can use any arbitrary Map function to consume records and produce key-value pairs as the intermediate output, and can use any arbitrary Reduce function to consume the intermediate output from the Map phase and produce the final output.

However, MapReduce applications do more than just apply these Map and Reduce functions to records and key-value pairs. For example, the output from the Map phase goes through several additional steps to decrease the amount of data that needs to be processed by the Reduce phase. After the Map function is applied to records, and key-value pairs are created, these pairs are sorted on their keys (Sort phase), and then combined to coalesce the values of duplicate keys (Combine phase).

This is important to do because it minimizes the amount of communication required between Map and Reduce phases, and it minimizes the amount of work that is done by the Reducer. Fortunately, these steps are often amenable to fixed-function acceleration, depending on the specific workload.

5.1 Map Phase Execution Breakdown

If applying the Map function to all input records was the dominant element of Mapper execution time, then accelerating Sort and Combine phases would be low-impact, and the focus should instead be placed on higher performance general purpose compute capability. However, for workloads that include Sort and Combine phases, Sort takes the largest portion of Mapper execution time, by a large margin, as seen in Figure 5.1. These results come from executing the Map phase on an NDC device.

Among these workloads, RangeAgg is the outlier, because it consists solely of a linear Map scan through memory, performing a range comparison on each record and conditionally incrementing a counter variable. There are no output key-value pairs from the Map phase, there is no Sort or Combine phases, and the Reduce phase is trivial (only adding together the output number from each Mapper), so we will not consider this workload a candidate for acceleration, and not consider it in the forthcoming results.

The other workloads' Map phases include the Map Scan phase, the Sort phase, the Combine phase, and the Partition phase. Of these phases, the Sort phase dominates execution time, taking from 84-94%. The Map Scan phase takes anywhere from 2-15%, but it requires general programmability, and cannot easily benefit from fixed function acceleration. Furthermore, although the Combine and Partition phases can benefit from fixed-function hardware acceleration, they take up such a small amount

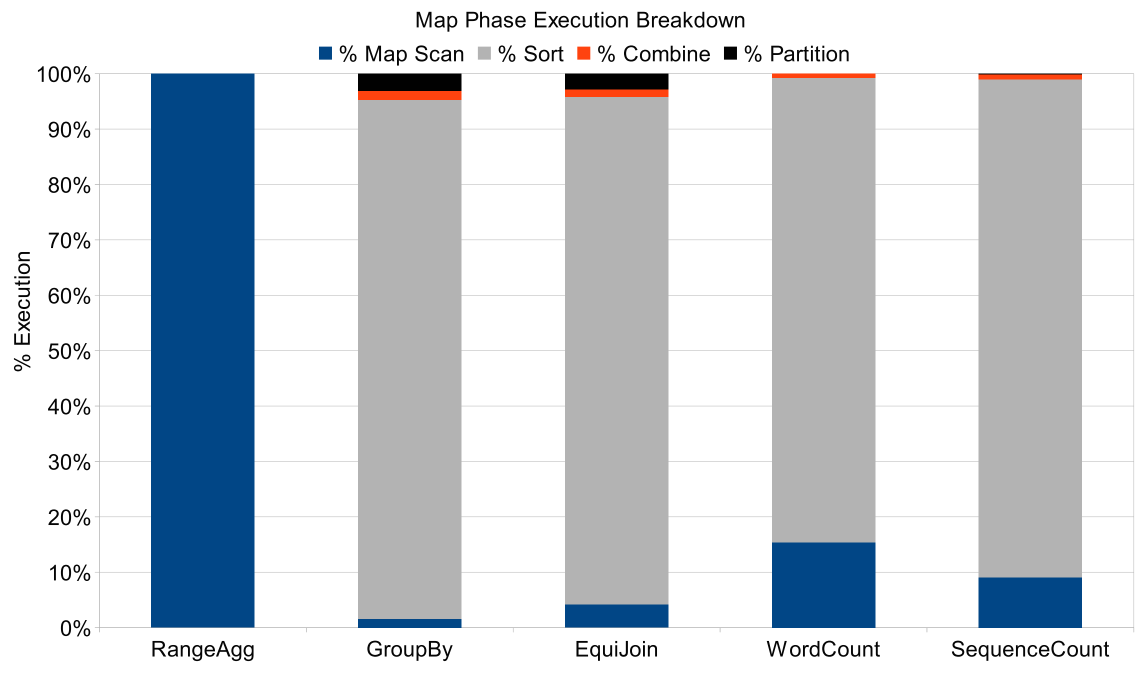


Figure 5.1. Execution breakdown of a Map task. Other than in RangeAgg, sorting dominates execution time.

of overall Map phase execution time that we do not focus our efforts on those phases. In these workloads, we are employing a simple hash-based partitioning method, but workloads that require range partitioning between Map and Reduce phases can further benefit from a hardware partitioning accelerator, like the one described by Wu et al. in [47].

First, we will discuss various hardware sorting accelerator strategies. Next, we will investigate how to integrate hardware sorting acceleration into various NDC implementations. Finally, we will evaluate the performance and energy impact of hardware sorting for NDC workloads.

5.2 Hardware Sorting Strategies

5.2.1 Sorting on Keys

The Sort phase in a MapReduce workload deals in arrays of key-value pairs, not just lists of integers. The key in a key-value pair is often an integer, but it may be something else, according to the domain of the currently executing workload. Even in instances where the key represents a floating point number, or even a non-number entity, it can still make sense to sort the keys as if they were integers. The purpose of the Sort phase it to create an ordering where all instances of a unique key appear consecutively. Treating all keys as integers, and sorting on those "integers" accomplishes this goal.

Sorting can even be done on keys that are longer than a single 32-bit integer word. In these instances, sorting will be done according to the first word in the key, and in the event of equality between two first words, the second words will be used as a tie-breaker, and the third words, and so on for however many words comprise the key.

5.2.2 Hardware Sort Algorithms

In this work, we will evaluate two different hardware sorting algorithms. First, we will look at a simple hardware implementation of a Merge Sort algorithm. Finally, we will look at a Bitonic sorter, and how that can be used to sort large data sets.

The hardware sorters all requires input and output stream buffers. The input stream buffers fetch all of the data between a low and high address. Whenever a cache line is consumed by a sorting unit, the input stream buffer will fetch the next cache line, and buffer it until it is used. Similarly, the output stream buffer writes the output key-value pairs to consecutive memory locations, starting at a given address.

5.2.3 Hardware Merge Sort

Merge sort works by interleaving the items from multiple sorted input lists into a larger sorted output list. At each step of the algorithm, the least of the items at the head of all the input lists is moved into the output list. The most common implementation of a merge sort merges two sorted input lists into a single sorted output list; however, the same principles apply when using more than two sorted input lists. In this work, we consider hardware merge sorters that merge just two input lists of key-value pairs, and eight input lists of key-value pairs.

The interface for the program to interact with both merge sorters is the same. The programmer must specify a base address for the input, a base address for the output buffer, the size of the key-value pairs (measured in 32-bit words), a bit mask for which words in the key-value pair should be used to perform comparisons to establish sorting order, and finally a number denoting which iteration of the merge sort algorithm the hardware merge sort unit should work on.

Merge sort is O(NlogN) in the worst case, scanning through the N items in the input logN times. In the first iteration, sorted lists of length 1 are merged into sorted lists of length 2, and in the second iteration, sorted lists of length 2 are merged into sorted lists of length 4, and so on until the entire list is sorted. This takes logN iterations to sort the entire input.

The programmer must set the iteration number when setting up the hardware merge sorter so that the accelerator knows how many sorted sublists have already been merged together, and what the memory access pattern will be to perform the next merging step. For example, if the programmer sets the iteration number to be 3 for the two-input merge sort accelerator, the accelerator will know that it should now merge sorted lists of length 4 into sorted lists of length 8 as it works through the input set.

We will next look at the differences between a hardware merge sorter that merges two sorted input lists, and a hardware merge sorter that merges eight sorted input lists. Both of these sorters can only produce a single sorted output per clock cycle, but they vary in the number of iterations required to sort a long list, and in their internal complexity and structure.

5.2.3.1 Two-Input Merge Sorter

First we consider a hardware merge sorter that merges two sorted input lists into a single sorted output list, as seen in Figure 5.2. In this implementation, the input stream buffers must keep track of two sorted input lists. On each cycle, the sorter compares its two input keys, then moves the smaller of the two key-value pairs onto its output. Once one of the two sorted input lists has been exhausted, the remaining list is chosen every cycle and copied to to the output.

This is more efficient than the software implementation of merge sort running on a general purpose core, because the general purpose core requires separate instructions for loading keys from each input list into registers, performing the comparison between them, and then copying the key-value pair with the lower key into the output buffer (even more loads and stores). This is on top of loop bounds-checking and branching overheads. The fixed-function hardware accelerator can perform in a single cycle what it takes many general purpose instructions to perform.

It takes a number of cycles equal to the number of key-value pairs sorted to complete one iteration of merge sort, and log base two of the number of key-value pairs to be sorted number of iterations to completely sort the entire data set.

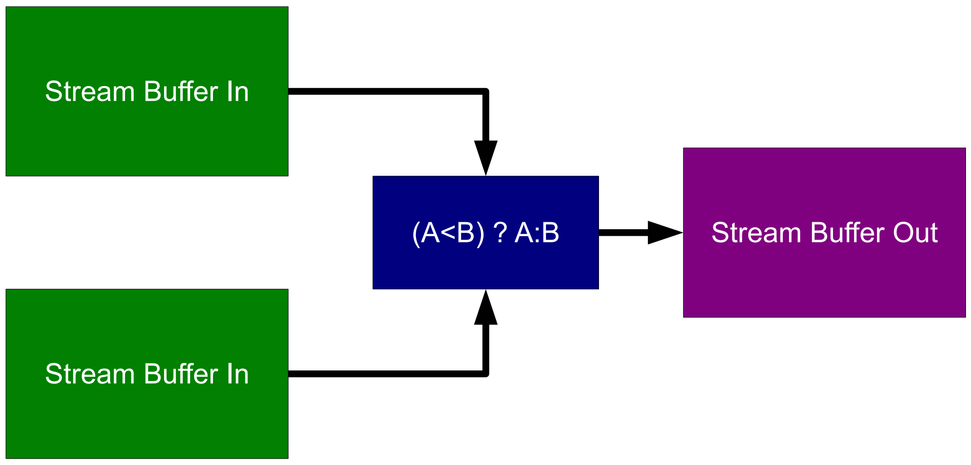


Figure 5.2. A Merge Sort unit that sorts two input lists.

5.2.3.2 Eight-Input Merge Sorter

Next we consider a hardware sorter that merges eight sorted input lists into a single sorted output list, as seen in Figure 5.3. In this implementation, the input stream buffers must keep track of eight sorted input lists. This sorter is internally organized like a heap, and each cycle, the smaller of two input key-value pairs will be chosen to advance to the next level, and over time, the lowest values will percolate toward the output. This means that each cycle, the smallest of all eight input lists will appear at the output, and be written to memory.

Similar to the two-input hardware merge sorter, each iteration of processing the input data set takes a number of cycle sequal to the number of key-value pairs being processed. The difference between the two is in the number of iterations required to completely sort the entire data set, which in this case is only log base eight of the number of input key-value pairs.

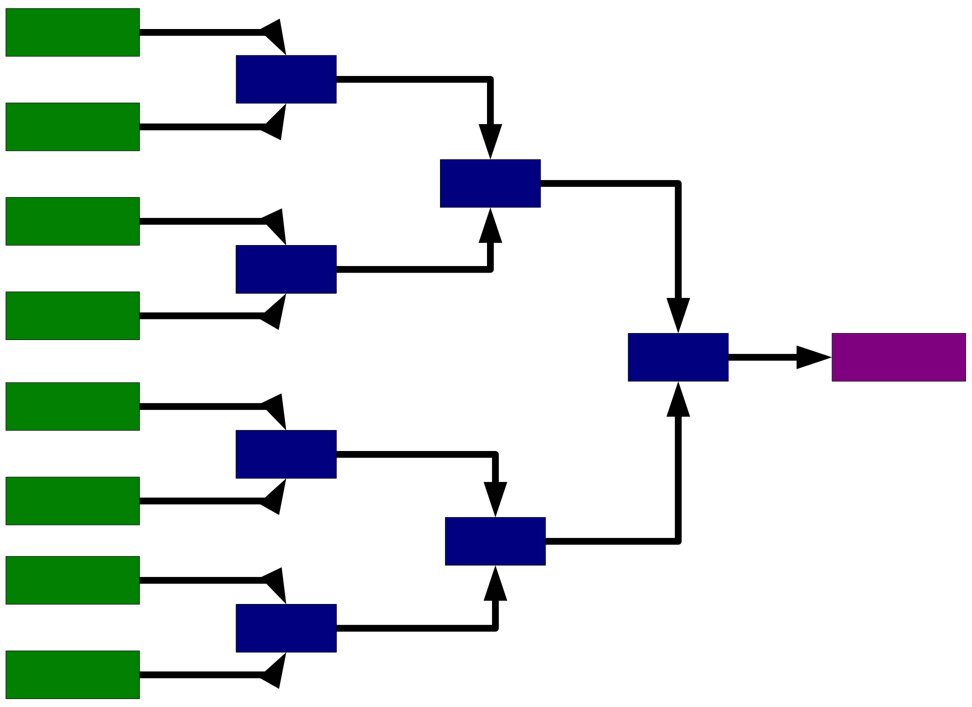


Figure 5.3. A Merge Sort unit that sorts eight input lists.

5.2.4 Hardware Bitonic Sort

Bitonic Sort is a sorting algorithm that first arranges the inputs in a bitonic ordering, and then afterwards quickly finishes sorting the inputs. Bitonic ordering means that half of the input set is monotonically increasing, and the other half is monotonically decreasing, or vice versa.

Bitonic sort is performed according to Figure 5.4. In that figure, the shaded areas represent stages of the sorting algorithm, and the arrows represent compares and swaps performed within a stage (the arrow points toward where the larger value will end up). All of these compares and swaps within a stage are independent of each other, and can all be performed on parallel hardware.

The stages of Bitonic sort can be grouped together, with each group of stages enforcing bitonic ordering on an ever-larger number of inputs. For example, the first group of stages is only one stage long, and it enforces bitonic ordering among adjacent pairs of inputs. The second group of stages is two stages long, and enforces bitonic ordering among sequences of four inputs. The third group of stages is three stages long, and enforces bitonic order among sequences of eight inputs, and so on.

The first (logN)-1 groups of stages establish the bitonic ordering in the entire input, and the final group of stages complete the final sorting. In total, there are logN groups of stages, and within each group of stages there are up to logN stages, so the entire algorithm is O(log squared N).

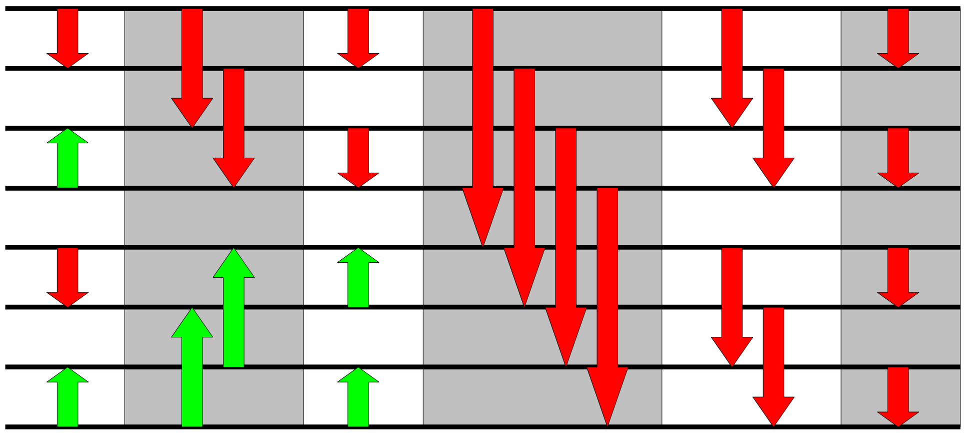


Figure 5.4. A Bitonic Sort unit that sorts eight input items.

5.2.4.1 Reliance on Other Sorting Mechanisms

A hardware implementation of Bitonic Sort can only operate on lists that are of a fixed, finite length. Each item that is going to be sorted must have storage within the sorter to hold it. For this reason, hardware sorters like a Bitonic Sorter work only on a fixed, often small, number of inputs. In this work, we consider a Bitonic Sort unit that can sort 1024 inputs, similar to Q100 [47]. Sorting inputs larger than that requires additional effort and other techniques.

In this work, we consider two other such techniques. In the first technique, Bitonic Sort is used as a finishing step to finalize the sorting that has been partially accomplished by a hardware range partitioning unit. This works by iteratively partitioning the inputs into smaller and smaller partitions until they are of size 1024 or smaller. At this point, the Bitonic Sort unit can quickly handle the remaining sorting steps. We consider a range partitioning unit that produces 256 output partitions on each iteration.

In the second technique, Bitonic Sort is used as a preprocessing step to prepare an input to be operated on by a hardware merge sorter. Using a 1024 input Bitonic Sorter bypasses ten iterations of merge sort when using a Merge Sorter with only two inputs, and bypasses three iterations of merge sort when using a Merge Sorter that takes eight inputs.

5.3 Hardware Sort Requirements

Fixed-function acceleration of database operations and hardware sorting were two key contributions of Q100 [47]. In that paper, the authors modeled various hardware accelerators in verilog, placed and routed their designs in 32 nm technology, and were able to produce area, power, and critical path latency figures for all units they considered. In this work, we use the figures they obtained for area, power, and critical path latency for the units we will use to build our hardware sorting algorithms, as seen in Table 5.1.

5.3.1 Two-Input Merge Sorter

We construct our two-input Merge Sorter out of a boolean generator pipelined into a pair of column filters. The boolean generator compares the keys of the two

Table 5.1. Parameters for Q100 units, used as building blocks to construct hardware sorting accelerators.

	Area	Power	Critical Path Lat.	Max Freq.
Boolean Generator	0.003 mm^2	0.2 mW	0.41 ns	2439 MHz
Column Filter	0.001 mm^2	0.1mW	0.23 ns	4348 MHz
Range Partitioner	0.942 mm^2	28.8 mW	3.17 ns	315 MHz
Bitonic Sorter	0.188 mm^2	39.4 mW	2.48 ns	403 MHz

inputs, and then produces a boolean value, which in turn controls the behavior of the two column filters. On each cycle, a new boolean value is produced, and on the following cycle it is consumed by the column filters, and only one of the column filters offers its key-value pair as output to the stream buffer. These two steps can be done in a pipelined manner.

According to the findings of the Q100 paper, a boolean generator has a critical path latency of 0.41 ns, for a maximum clock frequency of 2.4 GHz. Similarly, the column filters have a critical path latency of 0.23 ns, for a maximum clock frequency of 4.3 GHz. We find that even at much more modest clock speeds than these, the Merge Sorter is able to consume all available memory bandwidth, so we limit its clock speed to just 1 GHz, to match a fully programmable NDCore.

5.3.2 Eight-Input Merge Sorter

The eight-input Merge Sorter is constructed out of multiple two-input Merge Sorter units pipelined together in a tree fashion. We assume that the same clock speed can be reached with this sorter as with the two-input version, because the stages are pipelined, and that the power and area values scale linearly.

5.3.3 Bitonic Sort with Range Partitioner

Q100 used a Bitonic Sort unit in conjunction with a hardware Range Partitioner for all its sorting needs. First, the Range Partitioner iteratively creates smaller and smaller partitions until they are able to fit inside the Bitonic Sort unit. Then the Bitonic Sort unit consumes all of these partitions and sorts them. The total sorted output is all of the outputs of all of the Bitonic Sort operations stitched together.

According to the Q100 paper, a Range Partitioner has a critical path latency of 3.17 ns, for a maximum clock speed of 315 MHz, and a Bitonic Sorter has a critical path latency of 2.48 ns, for a maximum clock speed of 403 MHz. In this work, we round down both of those clockspeeds to the nearest 100 MHz, so the Range Partitioner operates at 300 MHz, and the Bitonic Sorter operates at 400 MHz.

In this work, we assume that optimal splitters are known beforehand with no additional overhead being modeled, but the current best known method for determining optimal splitters runs in O(k logN) time [63], where k is the number of desired partitions, and N is the number of input items. Furthermore, this method requires that the input already be sorted to begin with, defeating the purpose of using the Range Partitioner to aid with sorting in the first place.

5.3.4 Bitonic Sort with Merge Sorter

The Range Partitioner is able to accomplish its work in the above scheme in O(log base 256 N) iterations in the best case; however, this requires knowing beforehand, or calculating, the optimum ranges on which to partition the input. In the worst case, all inputs map to the same output partition, effectively doing zero useful work.

Merge sort, on the other hand, has identical best-, average-, and worst-case running times, with no preprocessing required. For this reason, we also pair the Bitonic Sorter with the eight-input Merge Sorter. First, the Bitonic Sorter sorts chunks of 1024 inputs, and then the Merge Sorter begins by merging pairs of two sorted 1024 inputs, and then continues on with the merge sort algorithm normally after that. As above, the Bitonic Sorter will operate at 400 MHz, and the Merge Sorter will operate at 1 GHz.

5.3.5 Summary of Hardware Sorting Unit Characteristics

Table 5.2 shows area, power, clock speed, and sorting throughput figures for all hardware sorters considered in this work. The Merge Sort units are trivially small, even compared to our 0.51 mm^2 general purpose NDCores. The Range Partitioner, however, is approximately 1.85 x the size of an NDCore, and the Bitonic Sorter is approximately 37% the size of an NDCore.

Table 5.2. Hardware sort accelerator parameters.

	Area	Power	Frequency
MergeSort2	0.005 mm^2	1.2 mW	1.0 GHz
MergeSort8	0.035 mm^2	8.4 mW	1.0 GHz
BitonicSort+RP	1.130 mm^2	81.3 mW	400 MHz (BS), 300 MHz (RP)
BitonicSort+MS8	0.223 mm^2	60.9 mW	400 MHz (BS), 1.0 GHz (MS)

The main advantage of using Bitonic Sort and a Range Partitioner is that it has the highest best-case performance, requiring the fewest iterations to sort data sets of size 128 MB and smaller, as in the workloads we examine. This best-case performance comes at the cost of using more die area, and having a much lower worst-case performance. Calculating optimal splitters for the Range Partitioner to use is infeasible in this sorting application, and merely approximating them takes processing time, and may still lead to worst-case performance.

The main advantage of using the MergeSort8 accelerator is that it also requires relatively few iterations to sort data sets of 128 MB of size and smaller, but its largest advantages are its very small die area cost, and its very favorable worst-case performance.

Even though we model the Bitonic Sort and Range Partitioner combination without overheads which may cripple its performance in a real system, we show in the coming sections that even with this unrealistic benefit, the performance gap between it and the accelerators which use MergeSort8 instead is not very large. The simpler, but longer-running, Merge Sort-based mechanisms might be preferred above the other sorting methods simply for reasons of die area alone.

Finally, while we only evaluate MergeSort2 and MergeSort8, it is easy to imagine other hardware merge sorters that merge different numbers of input lists. For example, a MergeSort256 accelerator would require the same number of iterations as the 256 partition Range Partitioner to process the same input set. A MergeSort256 sorter would require at least 2.56 mm^2 of die area (approximately 2.7x the die area of our Range Partitioner, and 73.1x the die area of the MergeSort8 accelerator), but due to the logarithmic nature of the merge sort algorithm (doubling of sorting resources yields a constant increase in performance), the limited performance benefit might not justify the increase in die area.

5.4 Integrating Hardware Sorting into NDC

Integrating fixed-function hardware accelerators into Near Data Computing requires changes both to hardware and to the way that software interfaces with the hardware. We return to the HMC-style NDC system described in Section 4.2.1 as our baseline system.

5.4.1 Hardware Changes

First, a hardware sorter unit is added to each existing NDCore in the system, now called a NDNode, for Near Data Node. This hardware sorter sits alongside the NDCore, and requires access to memory. Since hardware sorting units do not benefit from caches, but do benefit from prefetched streams, we also add streaming buffers to each NDNode. A complete NDNode can be seen in Figure 5.5.

The streaming buffers are the only connection that the hardware sorting unit has to memory. In order to not require hardware coherence between the NDCore cache and the streaming buffers, we instead flush the cache before hardware sorting can

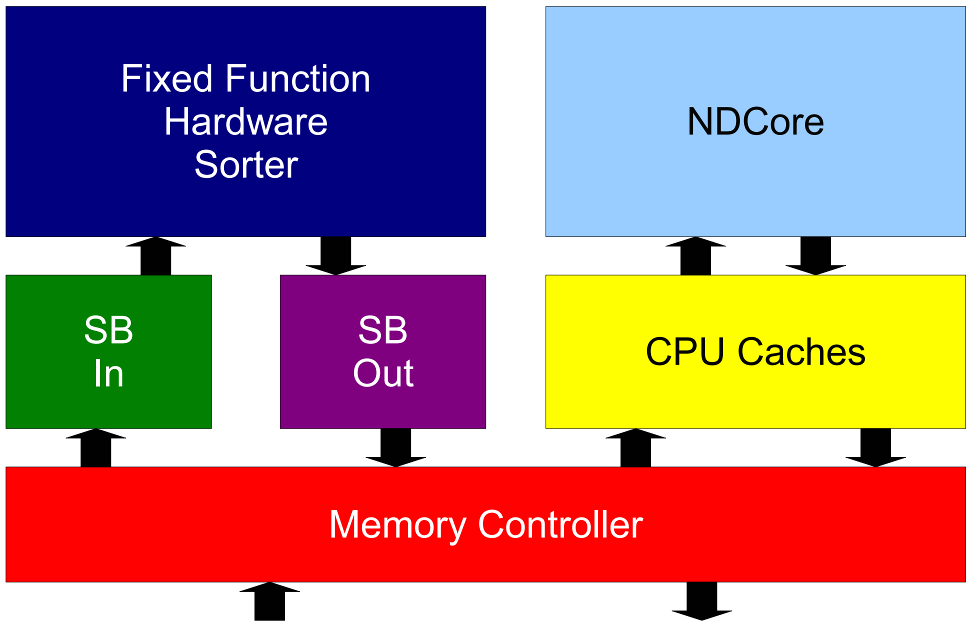


Figure 5.5. An NDNode consists of an NDCore, a hardware sorting accelerator, and streaming buffers.

commence, and ensure that the streaming buffer out has been completely drained before a hardware sorting operation is considered finished.

We refer to the work on the HARP range partition accelerator [46] for streaming buffer area and power figures. According to that paper, streaming buffers that are approproate for this task take up 0.13 mm^2 of die area at 32 nm, and consume 100 mW of power. We use these same figures for stream buffer power for all evaluated hardware sorters.

5.4.2 Software Changes

In order to interface with the hardware sort accelerator and streaming buffers, the NDCore can set and check the status of various registers. To set up a sorting task, the NDCore must program in the size of a record, as well as the size and location within the record of the key on which the sorting is performed. The NDCore must also program in which streaming buffers to use for this task, and this can be changed mid-sort if, for example, one of the input lists for a merge sort is completely consumed. After these are all set up, the hardware sorting unit will be available to begin sorting, as soon as a register bit to begin sorting is set, and the stream buffers are populated.

All of the particular details of interfacing with the registers can be handled by the MapReduce runtime, because it will know the structure of the lists of key-value pairs and how they should be sorted. The programmer therefore only needs to call a Sort() function on the output of the Map Scan phase, and the runtime can handle the rest.

As in the HARP work, we consider here stream buffers that are software-controlled. It is the job of the NDCore to monitor the status of how full the streaming buffers are, and fill the input streaming buffers and drain the output streaming buffers appropriately. The fill and drain access granularity is large enough that an NDCore operating at 1 GHZ can keep up with the throughput of the hardware sorters. When the input streaming buffers that it has been programmed to use are all populated, and the register bit to begin sorting has been set, the hardware sorting unit will continue until either of those conditions change.

5.5 Evaluating Hardware Sorting in NDC

5.5.1 Methodology

We use a similar methodology to the previous chapters to evluate hardware sorting accelerators in the context of Near Data Computing. We evaluate the same workloads, and our baseline system is the same system from Section 4.2.1. We augment this system with the various hardware sorting units described in Table 5.2. We model here an NDC HMC Mesh system, with full SerDes links on the logic layer of the NDC device. The primary change from the previous methodology is that we factor out the time and bandwidth the NDCore spends on sorting, and instead drop in a model based on the throughput and performance of the various hardware sorters. We also do not consider the RangeAgg workload, as it does not have Sort, Combine, and Partition phases, and focus instead on the other four from previous chapters.

We do not model the details of the NDCore filling and draining the streaming buffers, and we do not model the details of key-value pairs flowing through the hardware sorters. Instead, we assume that both NDCores and hardware accelerators are working at full utilization, and full power, during the sorting phase, even though the NDCores are only monitoring and maintaining the filling and draining of the streaming buffers.

Each NDNode in the system has access to a private vault of memory, and does not share resources with any other NDNode when using it. This means that all of the bandwidth of each vault can be dedicated to a single hardware sorting unit. We use the internal bandwidth assumptions from earlier chapters that each vault has a 32-bit internal data bus operating at 1600 Mbps per bus wire, for a maximum bandwidth of 6400 MB/s. Although sorting has total row buffer locality for both reads and writes, the fact that it performs as many reads as writes means that the data bus will need to be frequently turned around, incuring a small performance penalty each time. We therefore assume an 80% data bus utilization rate, despite the high row buffer locality, for a total per-vault bandwidth of 5120 MB/s combined read and write.

Although the BitonicSort + RangePartitioner combination theoretically can have the best performance, due to it dividing the input space into 256 outputs on each iteration, it is unlikely to do this perfectly in practice without analyzing the input set beforehand. However, we conservatively assume that it is able to divide the input into equal output partitions every time, and then show that this does not offer a very large advantage in practice compared to the hardware sorters using the MergeSort8 accelerator.

5.5.2 Performance Evaluation

5.5.2.1 Sorter Data Transfer Requirements

During the initial Map Scan phase of a Map task, records are read in as input, and key-value pairs are produced as output. The number and size of these key-value pairs are what determines sorting performance. Table 5.3 shows the number of key-value pairs output by the Map Scan of each of the evaluated workloads, the size of the key-value pairs, and the number of iterations that each sorting method requires to complete the sorting of those key-value pairs.

Each "iteration" in this context requires reading in the entire unsorted or partially sorted input from memory, processing it in the accelerator unit, and then writing it all back out to memory again. Lower iteration counts linearly decrease sorting execution time, and the amount of data read from and written to memory.

Using the MergeSort2 sorter offers no benefit over the software merge sorter when it comes to bytes transferred to and from memory, as seen in Figure 5.6. Only the sorters with lower algorithmic complexity are able to reduce the number of bytes transferred. This does not, however, mean that there is no advantage to the MergeSort2 sorter, as we will see when we look at Sort task latency.

Table 5.3. Output characteristics of each workload, and the number of sorting scans through memory needed for MergeSort2, MergeSort8, BitonicSort+RangePartition, and BitonicSort+MergeSort8.

	KV Size	KV Count	MS2	MS8	BS+RP	BS+MS8
RangeAgg	0 B	0	0	0	0	0
GroupBy	8 B	3.4 M	22	8	3	5
EquiJoin	12 B	3.4 M	22	8	3	5
WordCount	8 B	25.0 M	25	9	3	6
SequenceCount	16 B	25.0 M	25	9	3	6

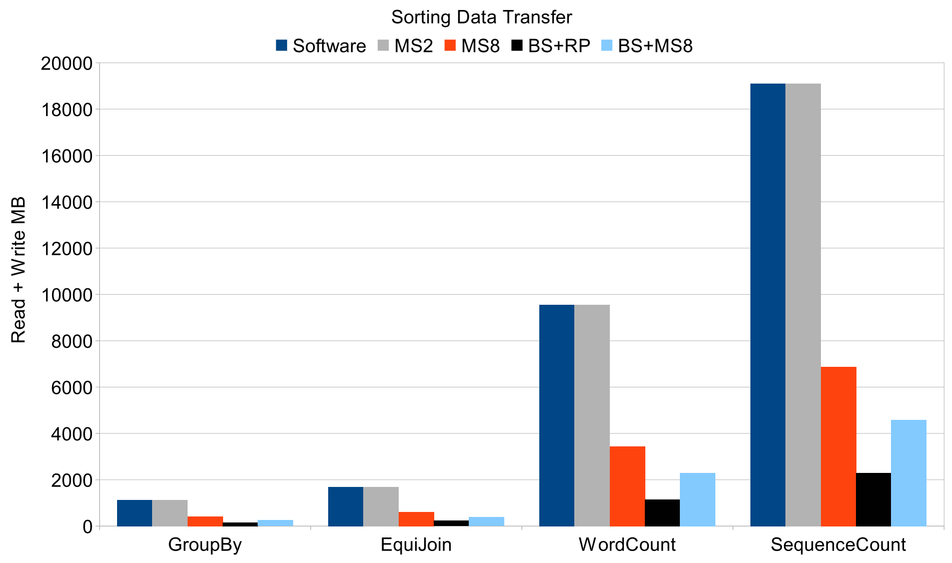


Figure 5.6. The number of bytes read and written to accomplish all iterations of sorting for each workload and hardware sorter.

5.5.2.2 Sorting Latency

As we have seen in previous chapters, the Map phase of a MapReduce workload takes the majority of execution time, and is the most important to accelerate. Furthermore, as we have seen in this chapter, Sort takes the majority of execution time within a Map phase, and it is therefore the most important to accelerate. We now compare the execution latency of running a software implementation of merge sort on an NDCore with the various accelerators described in this chapter, as seen in Figure 5.7.

Even though the MergeSort2 mechanism was not able to reduce the amount of data written to and from the DRAM, it is able to perform the sorting at such a higher rate than the NDCore and its software implementation, that it improves execution latency by an order of magnitude. The BitonicSorter and idealized RangePartition combination is able to improve upon the performance of MergeSort2 by another order of magnitude.

The two mechanisms that use MergeSort8, whether by itself or in concert with a BitonicSorter, achieve sorting performance in between MergeSort2 and the BitonicSorter and idealized RangePartition combination. Figure 5.7 shows the best-, average-, and worst-case performance for these MergeSort8-based sorters, but only the

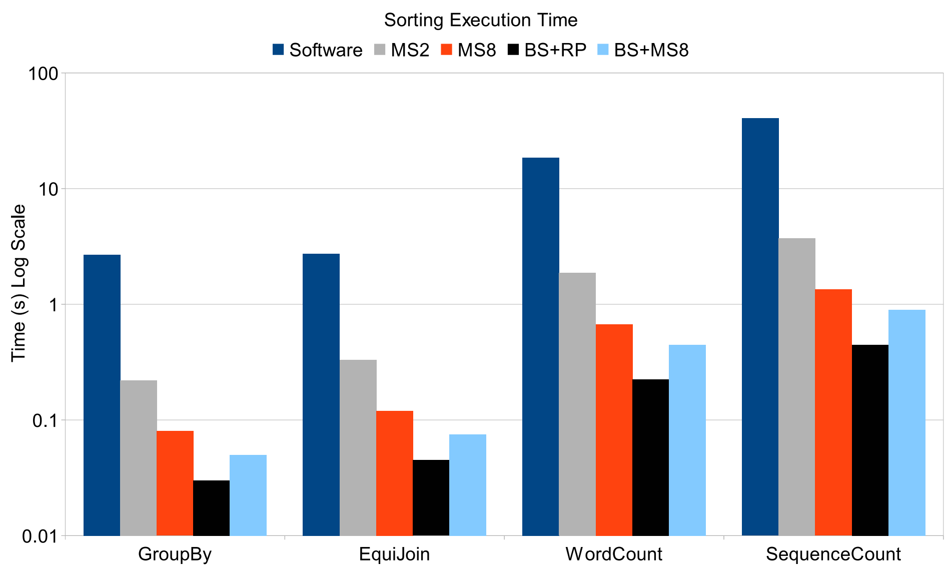


Figure 5.7. The number of seconds each sorting mechanism takes to accomplish the Sort phase for each workload. Note the logarithmic scale.

best-case performance for the RangePartitioner-based sorter. The RangePartition-based sorter may have much worse performance in reality.

5.5.2.3 Map Task Latency

Although the highest performing hardware sorter was able to offer two orders of magnitude performance over the software case handling the Sort phase of a Map task, that does not directly translate into overall system performance. Figure 5.8 shows the breakdown of the Map phase as executed first with software running on NDC, and then with the various accelerators.

After sort has been accelerated so much, it eventually ceases to be a substantial bottleneck in the system, and improving it further offers negligible additional gains. Workloads with more compute-intensive Map Scan phases have less sensitivity to variances in Sort phase performance, as seen in the cases of WordCount and SequenceCount.

For WordCount, the basic MergeSort2 accelerator is able to reduce sort execution latency by 90%, and the higher-performing BitonicSort and RangePartitioner combination is able to reduce sort execution latency by 98%, nearly another order of magnitude of sort performance. However, when considering the Map phase as a whole,

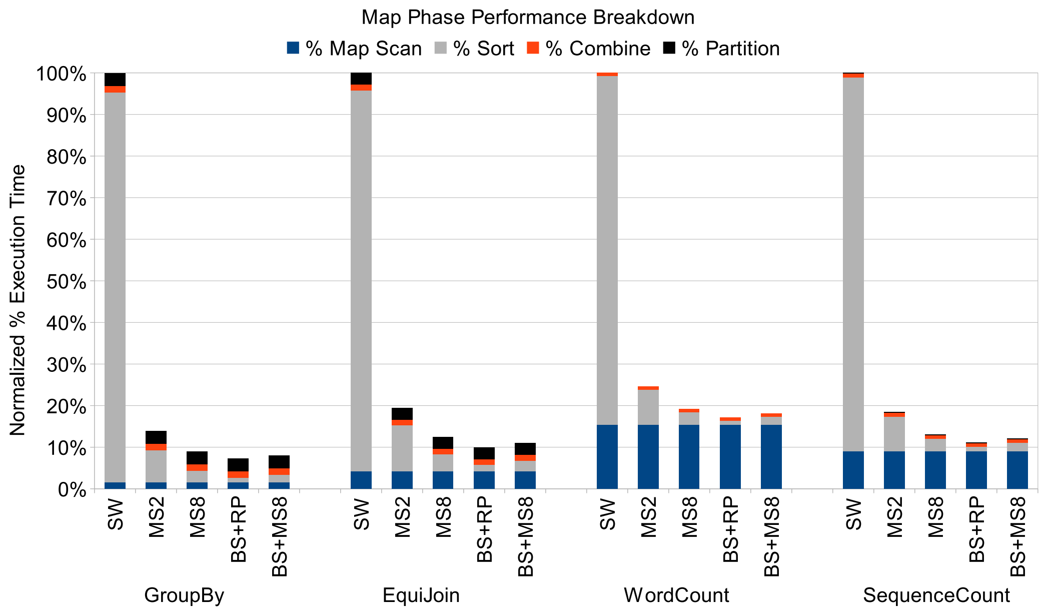


Figure 5.8. The amount of time spent in each phase of a Map task, normalized to the software case running on NDC.

MergeSort2 reduces execution latency by 75%, and the BitonicSort combination is able to reduce execution latency by only 83%, far short of the near order of magnitude difference when looking only at sort execution latencies.

MergeSort8 is even closer in performance to the BitonicSort combination, at 81% Map phase execution latency reduction for WordCount. Given its area and power advantages over the BitonicSort combinations, this might make for a strong case in favor of choosing it.

5.5.2.4 Sorting Compute Energy

We next look at the compute energy required to perform the Sort phase, ignoring memory energy for now. Compute energy for each system includes an 80 mW NDCore running for the duration of the Sort phase to feed the stream buffers, as well as 100 mW worth of stream buffers, and the hardware accelerators themselves running at full power for the duration. The breakdown for where energy is spent is shown in Figure 5.9.

We have conservatively assumed that the stream buffers and NDCore must be completely active at all times when a hardware sort accelerator is operating; however,

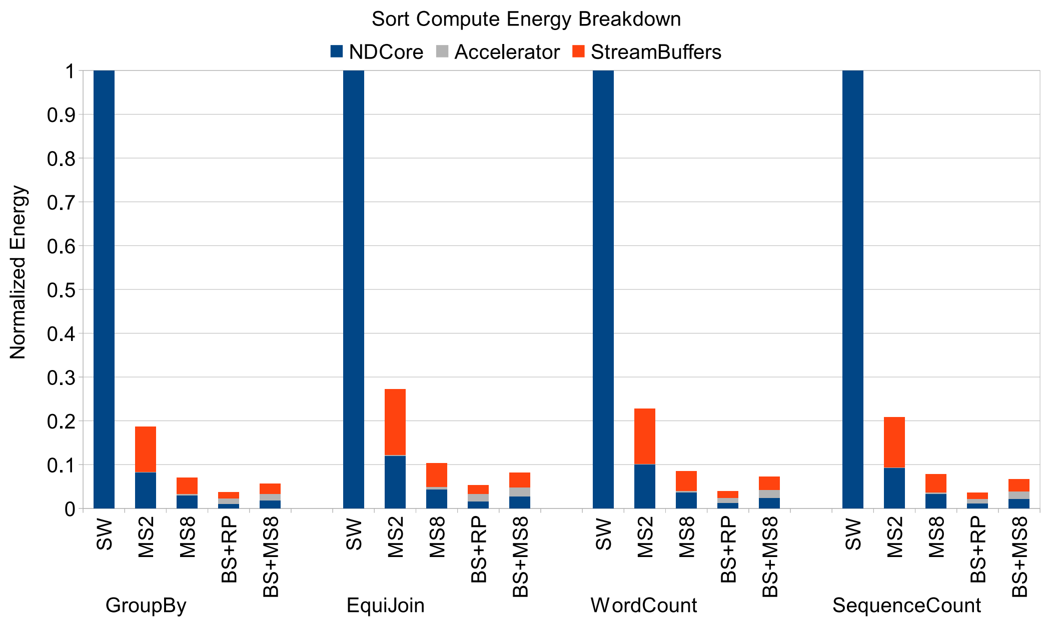


Figure 5.9. Breakdown of where compute energy is spent in the Sort phase, normalized to the NDCore Software case.

this may not be accurate. For the current assumptions, the energy used by the accelerator itself ranges from trivially small for the MergeSort accelerators, to still being only a fraction of the total energy spent by compute units during the Sort phase, as in the case of the two larger and more complex hardware sorters. NDCore and StreamBuffer energy is so high for MergeSort2 because it takes so much longer than the other accelerators to complete.

Although it appears that the energy consumption difference is substantial between the different hardware sort accelerators, we will next show that the energy consumed by compute units during the Sort phase is a small fraction of the overall energy spent, especially when compared to the energy consumed by the memory system.

5.5.2.5 Map Task Energy

We also consider total energy consumption of the memory and compute systems during Map phase execution, including both the Sort phase, and non-Sort phases. Figure 5.10 shows the breakdown for each workload and hardware sort accelerator, normalized to the case of the Software sort running on an NDCore.

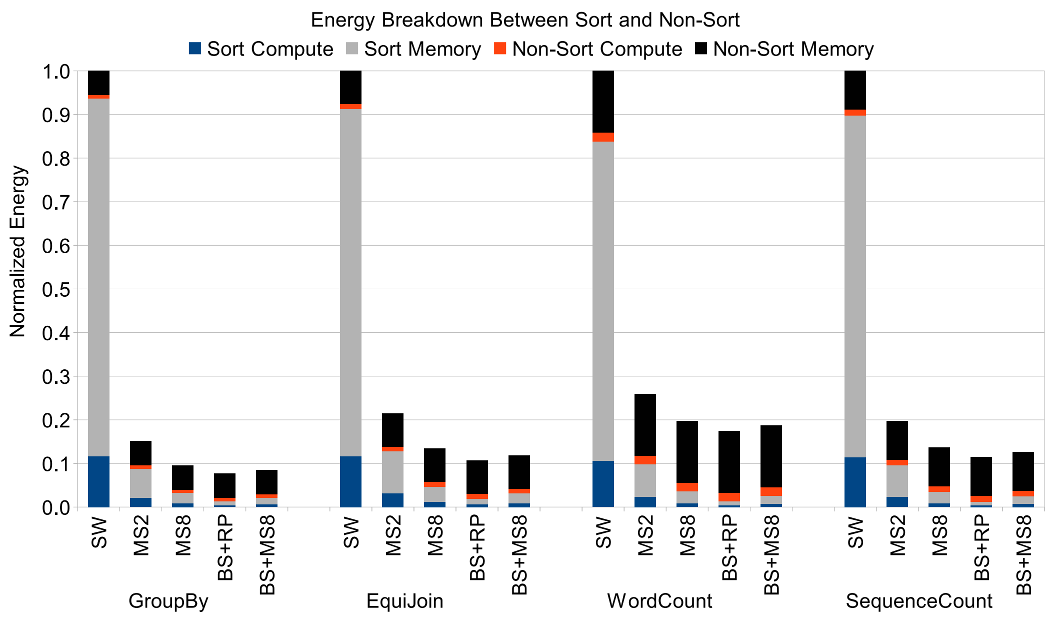


Figure 5.10. Breakdown of where both compute and memory energy is spent during Map phase execution, normalized to the NDCore Software case.

As with many other metrics we have examined so far, there is a large gap between the Software implementation of Sort and the MergeSort2 hardware accelerator. This gap ranges in size from a 74% energy consumption reduction in WordCount, to an 85% energy consumption reduction in GroupBy. There is another gap between MergeSort2 and the other hardware sort accelerators, which are all grouped fairly close to one another. The best of these, the BitonicSort and idealized RangePartitioner combination, sees energy conumption reduction from 82% in WordCount to 92% in GroupBy.

Looking at the energy consumption breakdown, it is clear that most of the energy is spent in the memory subsystem, and not in compute. All of the compute is being performed by low-EPI cores or fixed-function hardware accelerators, which offer large energy savings compared to conventional out-of-order server CPUs.

The memory, on the other hand, consumes most of the energy, using up to 7x more energy in the Software case, and up to 5x more energy in the hardware accelerator cases, than the compute resources. However, most of this energy consumption is due to high background power of HMC-based devices, like the NDC devices we are considering here. The SerDes links of the HMC are still responsible for the bulk of the energy consumption, and not DRAM array accesses.

In earlier chapters, we dismissed the idea of dynamically turning on and off SerDes links, because of their long wake up times. In the context of NDC devices equipped with hardware sort accelerators, the idea of turning SerDes links on and off may be even less feasible than it was before, because Map phase execution is being reduced so much that there is less time between Map and Reduce phases over which to amortize the cost of turning SerDes links on and off.

5.5.2.6 Sorting Accelerators with NDC-Module

Finally, we look at combining fixed-function hardware sorting accelerators with the NDC-Module system, described in Chapter 4. Each NDC-Module node has 4 sorting accelerators, one to go along with each of the node's 4 cores. We consider Map phase performance and energy consumption, as seen in Figure 5.11 and Figure 5.12, respectively. First we will review the performance and energy characteristics of the

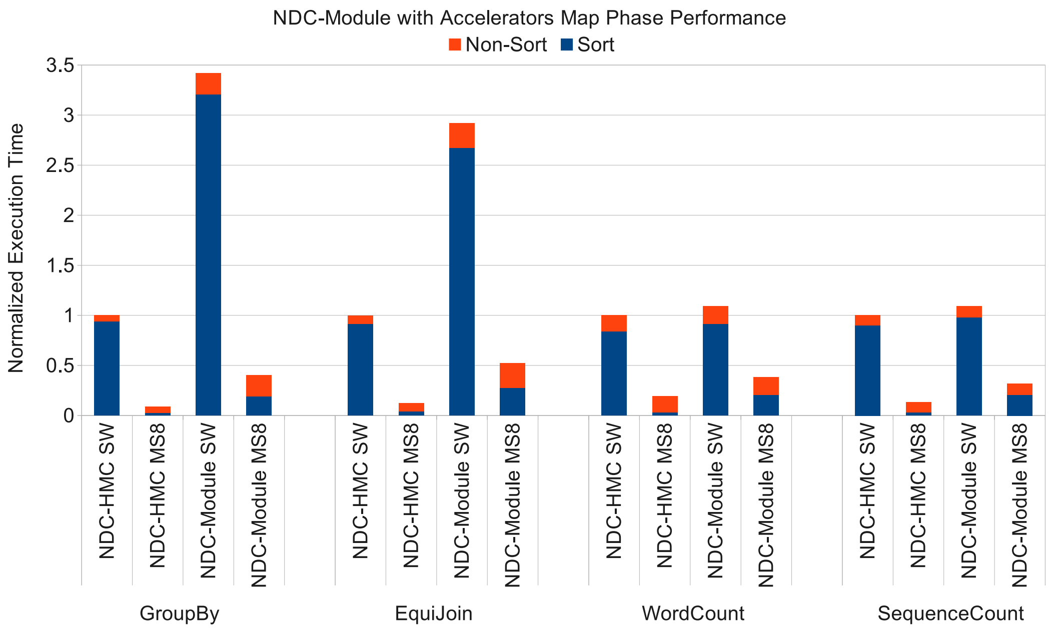


Figure 5.11. The breakdown of Map phase performance when using fixed-function hardware sorting accelerators in conjunction with the NDC-Module system, normalized to the NDC-HMC system.

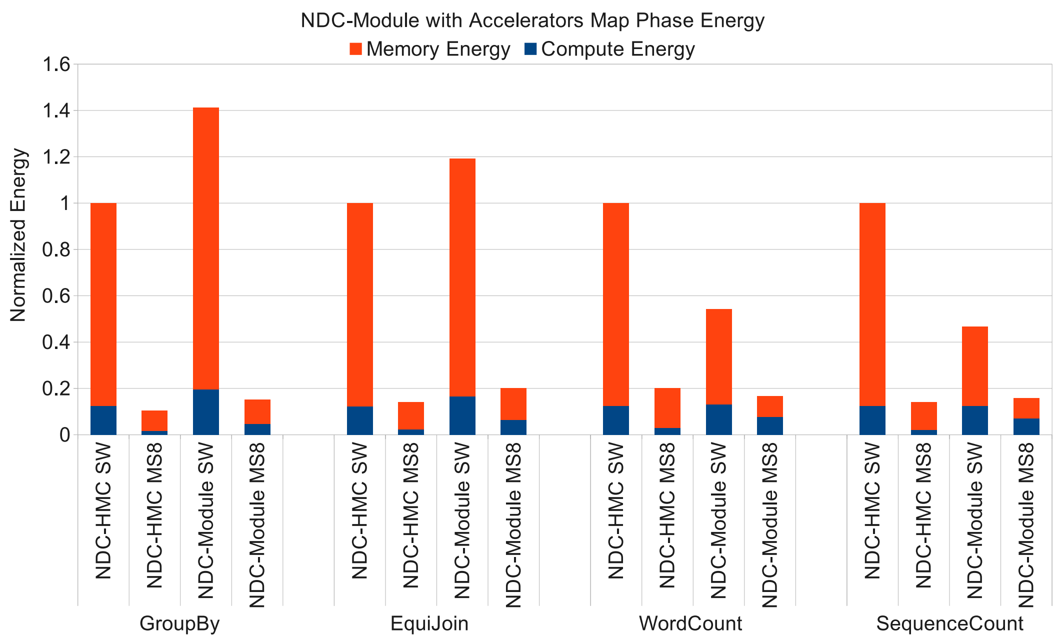


Figure 5.12. The breakdown of energy consumption between compute and memory sources when using fixed-function hardware sorting accelerators in conjunction with the NDC-Module system, normalized to the NDC-HMC system.

NDC-HMC and NDC-Module systems when running purely software implementations of our workloads.

In the software versions of GroupBy and EquiJoin, there is a large gap between the performance of the HMC-based NDC system, and the Module-based NDC system. That is because these workloads are bandwidth-bound, and the HMC-based NDC system offers much more bandwidth per core. In these workloads, NDC-HMC delivers 2.9x-3.4x the performance of the NDC-Module system. WordCount and Sequence-Count are more computation-bound, accessing memory far less than GroupBy and EquiJoin, and therefore performance does not suffer as greatly for the NDC-Module system running these workloads.

While NDC-HMC is strictly superior to NDC-Module from a performance perspective, things are different when we consider task energy consumption. In the two bandwidth-bound workloads, NDC-HMC is able to get the most out of its high power HMC memory system, and complete the task more quickly, saving energy overall. In the computation-bound workloads, NDC-Module is able to save energy on the memory side by using LPDDR2, which has very low idle power compared to HMC, and save on overall task energy.

The NDC-Module's memory system is often used at maximum bandwidth in the software implementations of GroupBy and EquiJoin, but WordCount and SequenceCount never approach those limits. However, fixed-function hardware sorting accelerators have the capacity to use all available memory bandwidth in the NDC-Module system in all workloads, during the sorting phase.

Because each accelerator in the NDC-Module system has less bandwidth to work with than the NDC-HMC system, the NDC-Module system takes on average 6.9x longer to complete the Sort phase, when both systems are using accelerators. Compared to NDC-Module without accelerators, using accelerators reduces Map phase execution time from 64.8% in WordCount up to 88.1% in GroupBy. In all cases, NDC-Module *with* sorting acceleration is able to improve upon the performance of NDC-HMC *without* sorting acceleration by at least a factor of 2x.

While the NDC-Module system was able to save 45%-54% on overall task energy compared to the NDC-HMC system when running the software version of the WordCount and SequenceCount workloads, using accelerators diminishes this advantage. In WordCount, NDC-Module with sorting acceleration consumes 14.4% less Map task energy than NDC-HMC with sorting acceleration, and in SequenceCount, NDC-Module with sorting acceleration consumes 14.6% more task energy than NDC-HMC with sorting acceleration. In all cases, the NDC-Module system *with* sorting acceleration is able to achieve greater energy efficiency than the NDC-HMC system *without* sorting acceleration by at least a factor of 5x.

5.6 Chapter Conclusions

In this chapter, we have explored the possibility of using fixed-function hardware accelerators as a way to speed up Map phase execution times. MapReduce workloads require general programmability, because of the Map and Reduce functions that the user supplies to the MapReduce system. The user-supplied Map function is applied during the Map Scan phase to all of the records in the input set, and intermediate key-value pairs are produced, which ultimately are consumed by the Reduce phase. The Reduce phase applies a user-supplied Reduce function to all of the intermediate key-value pairs produced by the Map phase. In between these phases are opportunities for fixed-function hardware accelerators (that lack general programmability) to improve performance.

We identified the Sort phase as a prime opportunity for hardware acceleration, because it dominates Map phase execution time (and therefore MapReduce execution time), and sorting lends itself well to hardware acceleration. We considered hardware merge sorters, bitonic sorters, and range partitioners. Ultimately we found the performance difference between the various hardware sorters to be small, as they all offer very large speedups compared to the case where sort is running in software on an NDCore.

We found that using the combination of BitonicSort and an idealized RangePartitioner are the best in terms of performance and energy consumption, but this is in part due to our assumption that the RangePartitioner would be able to perfectly divide the input set into equal sized output sets every iteration. We found that even with this optimistic assumption, this combination offers little beyond the performance and energy consumption characteristics of the MergeSort8 sorter, which has the benefit of using far less die area and power, and having very good worst-case performance. Using a merge sorter that interleaves more inputs than 8 would yield diminishing returns, as seen when comparing the Map phase performance of MergeSort8 and the best performing hardware sorter.

Compared to an NDC HMC Mesh system, incorporating fixed-function hardware accelerators offered Map phase execution time reductions ranging from 75% in WordCount using the MergeSort2 accelerator, to 91% reduction in GroupBy using the MergeSort8 accelerator. Similary, energy usage by the compute and memory subsystems during Map phase execution was reduced from 74% in WordCount by MergeSort2, to 90% in GroupBy by MergeSort8.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this chapter, we summarize the contributions of this dissertation and discuss remaining challenges for Near Data Computing-accelerated MapReduce workoads, and further applications of NDC outside of MapReduce workloads.

6.1 Contributions

Although Near Data Computing had fallen out of favor with researchers for several years, Big Data applications such as MapReduce, along with emerging 3D-stacking manufacturing technologies, make a strong argument for it once again. We showed that there are large opportunities for improving performance and saving energy in MapReduce workloads when compute capability is integrated more closely with memory devices.

We now review and summarize the main contributions of this dissertation.

6.1.1 Near Data Computing with 3D-Stacked Memory+Logic Devices

In Chapter 3, we began by arguing that commodity servers are a poor fit for MapReduce workloads, and designed a baseline system with MapReduce in mind. We began by replacing the commodity main memory system with Hybrid Memory Cubes (HCMs), and then replaced the traditional out-of-order CPU with few cores, with an in-order CPU with many cores (dubbed the EECore baseline). This baseline system was equipped with much more memory bandwidth than a typical server today, but thanks to the throughput-oriented CPU, insufficient memory bandwidth still proved to be a performance bottleneck. We overcame this bottleneck by using Near Data Computing (NDC), and replacing the HMCs in the main memory with NDC devices, which are similar to HMCs, but integrate 16 low energy-per-instruction cores into the logic layer, which are used for executing the Map phase of MapReduce workloads (Reduce is still executed on a central host CPU). Each NDCore operates on a single input data split, with a 1:1 ratio, and all splits are processed in parallel. Compared to the EECore baseline, the NDC system is able to decrease MapReduce execution times by 12.3% to 93.2%, and to decrease energy consumed by memory and compute units by 28.2% to 92.9%.

6.1.2 Near Data Computing Using Commodity Memory

In Chapter 4, we revisited the NDC system design from Chapter 3, and modified it to no longer be reliant on a central host CPU. This change reintroduces the Shuffle phase back into NDC MapReduce workloads. We found that for the workloads tested, the data transmitted between Map and Reduce phases are very small, and does not significantly contribute to overall execution times when data are being shuffled using the high-speed SerDes links of NDC devices. This chapter also introduces the NDC-Module system, which attempts to reproduce the beneficial performance and energy characteristics of the NDC system without using customized and potentially expensive HMC-based NDC devices. Instead, the NDC-Module system uses a combination of commodity LPDDR2 x32 DRAM devices and simple low-EPI 4-core CPUs. This departs from the NDC model where each CPU has its own dedicated vault, and therefore channel, of memory, and instead each channel of memory is shared by 4 cores. However, it is consistent with the NDC goal that every DRAM device in the system can contribute its full bandwidth toward workload execution. Compared to the HMC-equipped EECore system, the NDC-Module system decreases MapReduce execution times by 16.5% to 55.5%, and reduces compute and memory energy consumption by 68.3% to 84.4%.

6.1.3 Near Data Computing and Fixed Function Accelerators

In Chapter 5, we analyzed Map phase execution, and found the Sort phase to take up a large fraction of the time, and therefore to be a candidate for acceleration. We investigated hardware accelerators that implement merge sort, and bitonic sort (in conjuction with a hardware range partitioner). In particular, we investigated the MergeSort2 accelerator, which merges two sorted lists into a single sorted list, the MergeSort8 accelerator, which merges eight sorted lists into a single sorted list, BitonicSort and RangePartitioner combination, which iteratively partitions the input list into 256 output lists until the partitions are small enough to fit inside the bitonic sorter, and finally BitonicSort and MergeSort8 combination, which starts off by performing bitonic sort on chunks of 1024 items, and then finishes sorting using the MergeSort8 accelerator. The merge sort-based accelerators consume far less die area and power compared to the bitonic sorter or range partitioner, and have the added benefit of not having to deal with the difficulty in correctly choosing good splitters for the partitioner. Despite assuming ideal performance for the RangePartitioner, and it therefore having very good Sort phase performance, we found that when considering the execution of the entire Map phase, the benefit of the BitonicSort and RangePartitioner combination was perhaps not great enough to offset its much higher area and power requirements. Compared to an HMC-based NDC system, the MergeSort2 accelerator was able to reduce Map phase execution time by up to 75%, and reduce energy consumption by up to 74%. Similarly, the MergeSort2 was able to reduce Map phase execution time by up to 91%, and reduce energy consumption by up to 90%. The benefit of these accelerators when compared to the EECore system is even greater, reducing execution times and energy consumed by the compute and memory by up to two orders of magnitude.

6.1.4 Contributions Discussion

Even compared to a system with a conventional CPU and memory channel architecture that has been optimized for MapReduce workloads (the EECore system), Near Data Computing using HMC-based NDC devices has great potential for improving performance and reducing energy consumption. While the price of HMC devices is currently not available to the public, it is expected to be very high, and this may argue for an NDC approach that uses commodity commodity memory, as in the NDC-Module work. Finally, fixed-function accelerators can offer another order of magnitude performance and energy consumption improvement over HMC-based

NDC, and may be able to offer a compelling enough reason to adopt the expensive memory.

6.2 Future Work

We have so far described how Near Data Computing can be used to improve the execution of MapReduce workloads. We next look at other opportunities for NDC both within and without the realm of MapReduce.

6.2.1 Near Data Computing and Iterative MapReduce

So far we have considered a "workload" a single Map and single Reduce phase. As mentioned earlier in subsection 1.1.3, Spark is a programming framework built on top of Hadoop, and breaks away from the conventional two stage MapReduce paradigm, where the output of one round of Map and Reduce can be fed directly into the Map phase of the next round. These intermediate outputs are referred to as Resilient Distributed Datasets (RDDs). In a conventional MapReduce system, it is possible to carefully layout data across nodes in a cluster (or vaults in an NDC system), to ensure good Map phase locality and high performance, but with iterative MapReduce, it is impossible to predict how RDDs will be produced and how well behaved they will be. It might be beneficial to migrate data between nodes between one Reduce and the next Map phase, and to even use a hardware accelerator to decide which data should end up where.

6.2.2 Accelerating General Purpose Map and Reduce Functions with Reconfigurable Pipes

Once we accelerated the Sort of Map execution, it diminished as a bottleneck, and instead the Map Scan phase arose as a new bottleneck. Unfortunately, the Map Scan phase is not as easy to accelerate as the Sort phase. While Map and Reduce functions do require general programmability from the programmer's perspective, this does not mean that the general code must compile to conventional instructions that are executed on conventional RISC cores in an NDC system. A Map or Reduce function can be translated into a pipelined dataflow plan through ALUs, registers, and muxes, and this can be mapped onto reconfigurable hardware in the core. This would allow fast, efficient, and low-overhead execution of arbitrary Map and Reduce functions, diminishing the new bottleneck that has arisen. The challenges for this would be devising a way to automatically translate between compiled code and a reconfigurable pipeline plan, and evaluating the amount of reconfigurable hardware required to express a large variety of Map and Reduce functions.

6.2.3 Near Data Computing and General Database Acceleration

In the Q100 work [47], the authors only considered a single Data Processing Unit (DPU) acting as either a standalone co-processor, or as an added co-processor working with a general purpose CPU. Furthermore, they required that their entire input data sets be processed by only one DPU, not coordinating with any other DPUs or CPUs. This is in contrast to the data-parallel approach taken in this dissertation, where many small splits of the input data set are each processed by independent compute units. There is an opportunity for the ideas of NDC and a Q100-like DPU to be combined into a system where each DRAM chip, containing a separate split of the input data set, is connected to an independent DPU, and all of these DPUs can work together in a data-parallel fashion to execute database operations.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of OSDI*, 2004.

[2] "The Apache Hadoop Project." http://hadoop.apache.org/.

[3] "Memcached: A Distributed Memory Object Caching System." http://memcached.org.

[4] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazieres, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM," *SIGOPS Operating Systems Review*, vol. 43(4), 2009.

[5] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of HotCloud*, 2010.

[6] SAS, "SAS In-Memory Analytics." http://www.sas.com/software/high-performance-analytics/in-memory-analytics/.

[7] SAP, "In-Memory Computing: SAP HANA." http://www.sap.com/solutions/technology/in-memory-computing-platform.

[8] BerkeleyDB, "Berkeley DB: high-performance embedded database for key/value data." http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html.

[9] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA Database – An Architecture Overview," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.

[10] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA Database: Data Management for Modern Business Applications," *SIGMOD Record*, vol. 40, no. 4, pp. 45–51, 2011.

[11] "Intel Xeon Processor E7-8890 v2 Specifications." http://ark.intel.com/products/75258.

[12] D. Kim *et al.*, "3D-MAPS: 3D Massively Parallel Processor with Stacked Memory," in *Proceedings of ISSCC*, 2012.

[13] G. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors," in *Proceedings of ISCA*, 2008.

[14] G. Loi, B. Agrawal, N. Srivastava, S. Lin, T. Sherwood, and K. Banerjee, "A Thermally-Aware Performance Analysis of Vertically Integrated (3-D) Processor-Memory Hierarchy," in *Proceedings of DAC-43*, June 2006.

[15] N. Madan, L. Zhao, N. Muralimanohar, A. N. Udipi, R. Balasubramonian, R. Iyer, S. Makineni, and D. Newell, "Optimizing Communication and Capacity in a 3D Stacked Reconfigurable Cache Hierarchy," in *Proceedings of HPCA*, 2009.

[16] X. Dong, Y. Xie, N. Muralimanohar, and N. Jouppi, "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support," in *Proceedings of SC*, 2010.

[17] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A Novel Architecture of the 3D Stacked MRAM L2 Cache for CMPs," in *Proceedings of HPCA*, 2009.

[18] D. H. Woo *et al.*, "An Optimized 3D-Stacked Memory Architecture by Exploiting Excessive, High-Density TSV Bandwidth," in *Proceedings of HPCA*, 2010.

[19] Samsung Electronics Corporation, "Samsung Electronics Develops World's First Eight-Die Multi-Chip Package for Multimedia Cell Phones," 2005. (Press release from `http://www.samsung.com`).

[20] Samsung, "Samsung to Release 3D Memory Modules with 50% Greater Density," 2010. http://www.computerworld.com/s/article/9200278/Samsung_to_release_3D_memory_modules_with_50_greater_density.

[21] Elpida Memory Inc., "News Release: Elpida Completes Development of Cu-TSV (Through Silicon Via) Multi-Layer 8-Gigabit DRAM." http://www.elpida.com/pdfs/pr/2009-08-27e.pdf, 2009.

[22] Elpida Memory Inc., "News Release: Elpida, PTI, and UMC Partner on 3D IC Integration Development for Advanced Technologies Including 28nm." http://www.elpida.com/en/news/2011/05-30.html, 2011.

[23] Tezzaron Semiconductor, "3D Stacked DRAM/Bi-STAR Overview," 2011. http://www.tezzaron.com/memory/Overview_3D_DRAM.htm.

[24] "Hybrid Memory Cube, Micron Technologies." http://www.micron.com/innovations/hmc.html.

[25] T. Pawlowski, "Hybrid Memory Cube (HMC)," in *HotChips*, 2011.

[26] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube – New DRAM Architecture Increases Density and Performance," in *Symposium on VLSI Technology*, 2012.

[27] "Micron DDR3 SDRAM Part MT41J256M8," 2006.

[28] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," in *Proceedings of SC*, 1999.

[29] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. Kang, I. Kim, and G. Daglikoca, "The Architecture of the DIVA Processing-In-Memory Chip," in *Proceedings of ICS*, 2002.

[30] J. Gebis, S. Williams, C. Kozyrakis, and D. Patterson, "VIRAM-1: A Media-Oriented Vector Processor with Embedded DRAM," in *Proceedings of DAC*, 2004.

[31] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and C. Yelick, "A Case for Intelligent DRAM: IRAM," *IEEE Micro*, vol. 17(2), April 1997.

[32] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. Yelick, "Intelligent RAM (IRAM): the Industrial Setting, Applications, and Architectures," in *Proceedings of ICCD*, 1997.

[33] J. Brockman, S. Thoziyoor, S. Kuntz, and P. Kogge, "A Low Cost, Multithreaded Processing-in-Memory System," in *Proceedings of WMPI*, 2004.

[34] R. Murphy, P. Kogge, and A. Rodrigues, "The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems," in *Proceedings of Workshop on Intelligent Memory Systems*, 2000.

[35] Y. Kang, M. Huang, S. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *Proceedings of ICCD*, 1999.

[36] P. Trancoso and J. Torrellas, "Exploiting Intelligent Memory for Database Workloads," in *Proceedings of Workshop on Memory Performance Issues*, 2001.

[37] J. Lee, Y. Solihin, and J. Torrellas, "Automatically Mapping Code on an Intelligent Memory Architecture," in *Proceedings of HPCA*, 2001.

[38] T. Sterling and H. Zima, "Gilgamesh: A Multithreaded Processor-in-Memory Architecture for Petaflops Computing," in *Proceedings of SC*, 2002.

[39] A. Gutierrez, M. Cieslak, B. Giridhar, R. Dreslinski, L. Ceze, and T. Mudge, "Integrated 3D-Stacked Server Designs for Increasing Physical Density of Key-Value Stores," in *Proceedings of ASPLOS*, 2014.

[40] A. Rodrigues, R. Murphy, R. Brightwell, and K. Underwood, "Enhancing NIC Performance for MPI using Processing-in-Memory," in *Proceedings of Workshop on Communication Architectures for Clusters*, 2005.

[41] M. Gokhale, B. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massively Parallel PIM Array," *IEEE Computer*, 1995.

[42] K. Mai, T. Paaske, N. Jaysena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *Proceedings of ISCA*, 2000.

[43] A. Saulsbury, F. Pong, and A. Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration," in *Proceedings of ISCA*, 1996.

[44] M. Oskin, F. Chong, and T. Sherwood, "Active Pages: A Model of Computation for Intelligent Memory," in *Proceedings of ISCA*, 1998.

[45] K. Lim, D. Meisner, A. Saidi, P. Ranganathan, and T. Wenisch, "Thin Servers with Smart Pipes: Designing Accelerators for Memcached," in *Proceedings of ISCA*, 2013.

[46] L. Wu, R. Barker, M. Kim, and K. Ross, "Navigating Big Data with High-Throughput Energy-Efficient Data Partitioning," in *Proceedings of ISCA*, 2013.

[47] L. Wu, A. Lottarini, T. Paine, M. Kim, and K. Ross, "Q100: The Architecture and Design of a Database Processing Unit," in *Proceedings of ASPLOS*, 2014.

[48] T. Farrell, "HMC Overview: A Revolutionary Approach to System Memory," 2012. Exhibit at Supercomputing.

[49] "Intel Xeon Processor E5-4650 Specifications." http://ark.intel.com/products/64622/.

[50] ITRS, "International Technology Roadmap for Semiconductors, 2009 Edition."

[51] G. Sandhu, "DRAM Scaling and Bandwidth Challenges," in *NSF Workshop on Emerging Technologies for Interconnects (WETI)*, 2012.

[52] O. Azizi, A. Mahesri, B. Lee, S. Patel, and M. Horowitz, "Energy-Performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis," in *Proceedings of ISCA*, 2010.

[53] "Cortex-A5 Processor." http://www.arm.com/products/processors/cortex-a/cortex-a5.php.

[54] S. Keckler, "Life After Dennard and How I Learned to Love the Picojoule." Keynote at MICRO, 2011.

[55] P. Kundu, "On-Die Interconnects for Next Generation CMPs," in *Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems (OCIN)*, 2006.

[56] J. Howard *et al.*, "A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS," in *Proceedings of ISSCC*, 2010.

[57] M. Arlitt and T. Jin, "1998 World Cup Web Site Access Logs." http://www.acm.org/sigcomm/ITA/, August 1998.

[58] "PUMA Benchmarks and dataset downloads." https://sites.google.com/site/farazahmad/pumadatasets.

[59] "Wind River Simics Full System Simulator." http://www.windriver.com/products/simics/.

[60] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: the Utah SImulated Memory Module," tech. rep., University of Utah, 2012. UUCS-12-002.

[61] "Micron System Power Calculator." http://www.micron.com/products/support/power-calc.

[62] "HotSpot 5.0." http://lava.cs.virginia.edu/HotSpot/index.htm.

[63] W. Le, F. Li, Y. Tao, and R. Christensen, "Optimal Splitters for Temporal and Multi-version Databases," in *Proceedings of SIGMOD*, 2013.