REVISITING THE MEMORY HIERARCHY IN THE MANY-CORE ERA: COMPUTATION IS CHEAP, BANDWIDTH IS EVERYTHING

by

Yamuna Rajasekhar

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical Engineering

Charlotte

2014

ABSTRACT

YAMUNA RAJASEKHAR. Revisiting the memory hierarchy in the many-core era: Computation is cheap, bandwidth is everything.
(Under the direction of DR. RONALD R. SASS)

Integrated Circuits (ICs) for logic (computation) have dramatically increased in both capacity *and* speed since their introduction in 1958. However, memory technology has had only modest improvements in speed. Moreover, to increase capacity and lower the cost per bit, main memory is implemented as a separate, external IC. Thus, relative to logic speeds, memory latency is increasing and physical constraints on external pins limits memory bandwidth. The traditional on-chip cache hierarchy has evolved with the sole goal hiding external memory latency for a single-core, sequential processor; essentially giving the illusion of lower latency. Unfortunately, it does so at the expense of IC resources (transistors), energy, power, and memory bandwidth. As the world quickly moves toward multi/many-core architectures, current cache architectures are not aligned with and in fact, are hostile to future priorities.

This dissertation questions the allocation of on-chip resources for logic ICs and proposes a novel memory architecture that (a) *actively* manages the movement of on-chip and off-chip data, (b) creates a flatter memory hierarchy, and (c) emphasizes efficient bandwidth utilization over latency. The results demonstrate that, when combined with a suitable programming environment, the proposed memory subsystem enables a larger fraction of chip resources to be dedicated to computation. This yields a higher degree of parallelism and ultimately reduces the time-to-completion of an application independent of how fast individual tasks execute.

ACKNOWLEDGMENTS

In my way of pursuing this doctoral degree, many people have supported and encouraged me in many different ways to make my endeavor a success. I feel extremely blessed to have been surrounded by adept professors, a fantastic family, and amazing friends and colleagues who have stood by me through it all. I would like to thank:

First and foremost, my parents, Rajee and Rajasekhar, who are the reason of every iota of success that I have achieved in my life. My super mom always kept me on track and invented countless ways to motivate me when I was down. My incredible dad is my biggest pillar of strength and I know I can count on him for anything, at any time. Thank you, ma and pa, for being stellar role models and I am lucky that you were my first teachers.

My marvelous brother, Raman, who always has a way of making things alright for me. You keep me grounded, but yet are often the wind beneath my wings.

My brilliant advisor, Dr. Ron Sass, for his guidance, encouragement and the wealth of knowledge he has imparted to me. My life as a graduate student was truly enriched because I had him as my advisor. My committee members, Dr. Jim Conrad, Dr. Bharat Joshi, and Dr. Jing Yang for their time, support, guidance and most importantly patience. My friends and colleagues at the RCS lab for all their help and support, especially Will and Robin.

My extraordinary husband, Sandeep, who is my best friend, the yang to my yin, my superhero. He has showered me with unconditional love and unwavering support through this long journey. His unparalleled patience, and his capacity to endure is what keeps us going. I really could not have done this without you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Over the history of computer architecture, Moore's law[1] [1] has been fundamental in defining and predicting computer hardware. Arguably, this has mainly been possible due to the Dennard scaling[2] theory [2], which implies that power density will remain constant. That is, the decrease in transistor size enables engineers to lower the voltage and, in turn, achieve a higher clock frequency without affecting the functionality. However, recent changes in computer architecture have used Moore's Law to improve performance by shifting resources from improving single core performance to implementing ensembles of parallel cores [3]. With this enormous shift in the conventional trend, Pollack's rule[3] [4], became significant. This has led to the successful investigation of multiple cores on a single chip operating at a lower speed and power but working together in parallel. Pollack's rule also predicts that microarchitecture advances will lead to a two-fold increase in the performance, while maintaining the same power consumption. This, in turn has greatly improved the performance of various subsystems in multi-core architectures.

It has been a different story for memory devices. Figure 1.2 shows an abbreviated history of computer memory. Starting with solid state memory, roughly at the end of the 1960s, memory performance has evolved steadily. In contrast to logic ICs, the number of transistors per chip continues to increase and networking speeds continue to improve, off-chip memory subsystems are not improving, and have in fact remained

[1]Moore's law states that the number of transistors in a dense integrated circuit doubles approximately every eighteen months.

[2]Dennard Scaling states that as transistors get smaller, their power density remains constant, so the power use is directly proportional to the area and both voltage and current scale downward.

[3]Pollack's rule states that performance increase due to microarchitecture advances is roughly proportional to the square root of the increase in complexity.

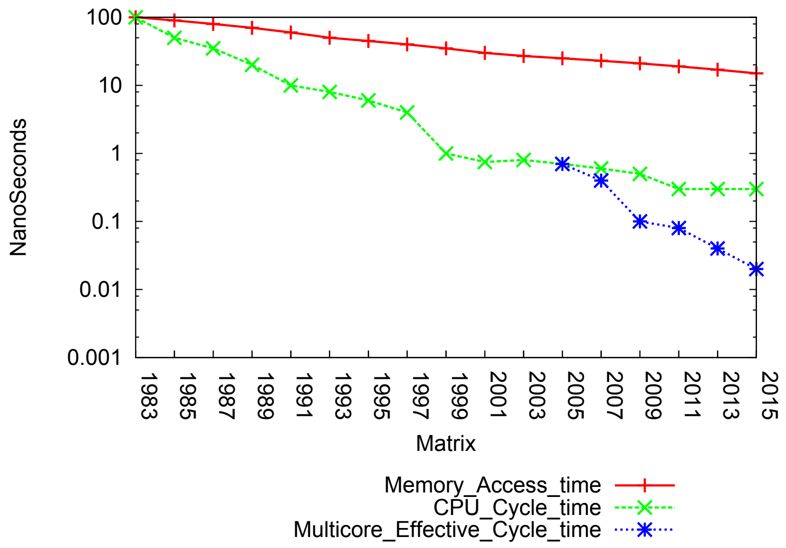


Figure 1.1: Memory latency compared to (processor) clock frequency

stagnant through the progression of single-core to multi-core architectures. Latencies to off-chip have remained virtually flat in the past three decades, as illustrated in Figure 1.1. Moreover, it is highly likely that the long-term trend in memory latency will continue for the next decade.

Prior to the multi/many-core era, caches were developed to address this issue, providing the processor the illusion of low latency. The classic single core–memory relationship, shown in Figure 1.3, uses a complicated cache hierarchy to hide latency to off-chip memory.

When measured in terms of system clock cycles, memory latency has *grown* dramatically and it is expected to approach 1,000 clock cycles to access off-chip DRAM in the next several decades [5]. This so called “Memory Wall” was recognized in the mid-1990s [6] but the extensive use of caches and cache hierarchies has allowed chip architects to mitigate the latency issue for single core microprocessors. This approach has come with a high cost. It requires a substantial amount of resources (from 60% to 90% of the transistors on a chip) [7]. It also consumes a significant amount of energy to operate due to leakage current and unnecessary data movement [8]. Additionally,

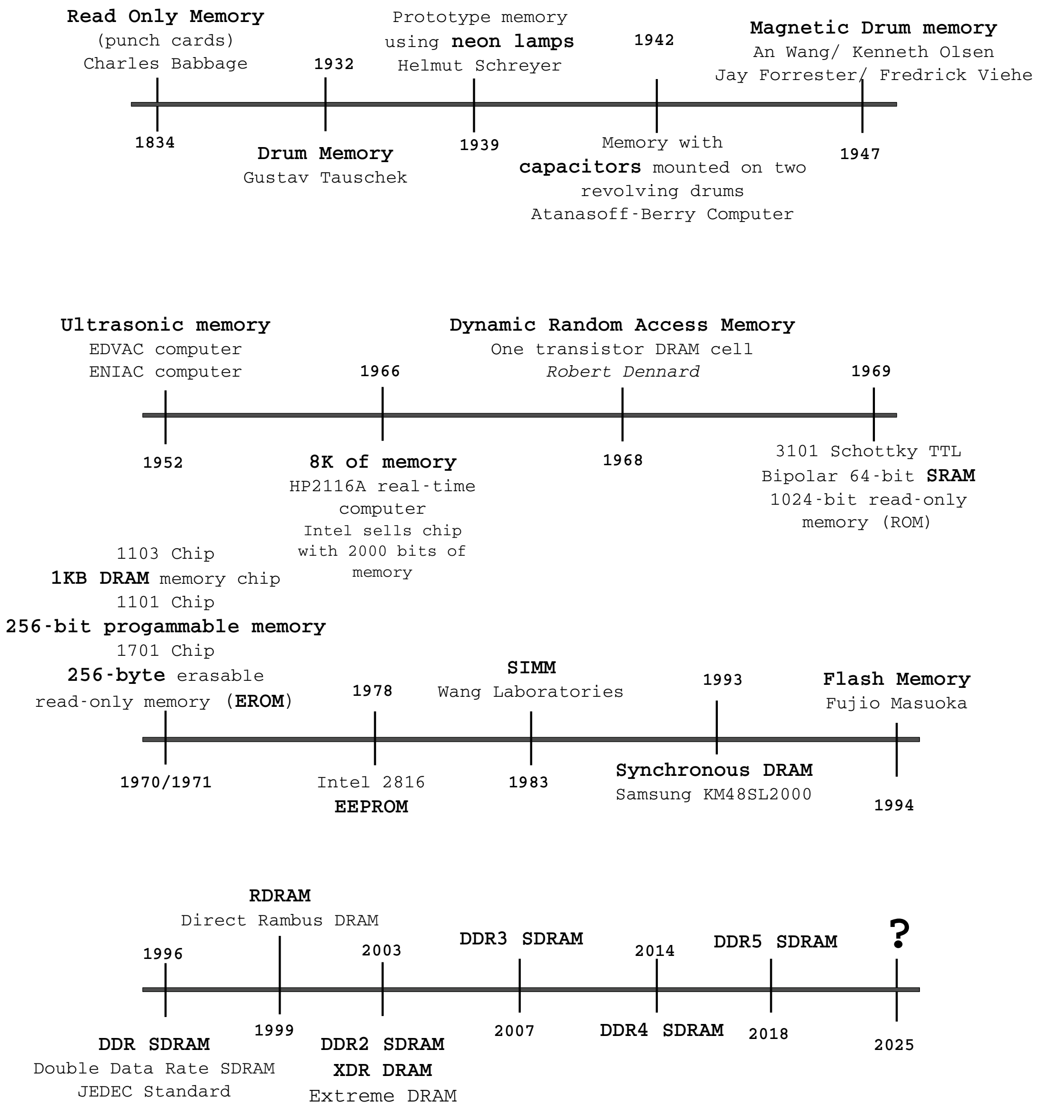


Figure 1.2: Timeline of Memory devices invented over the past few decades

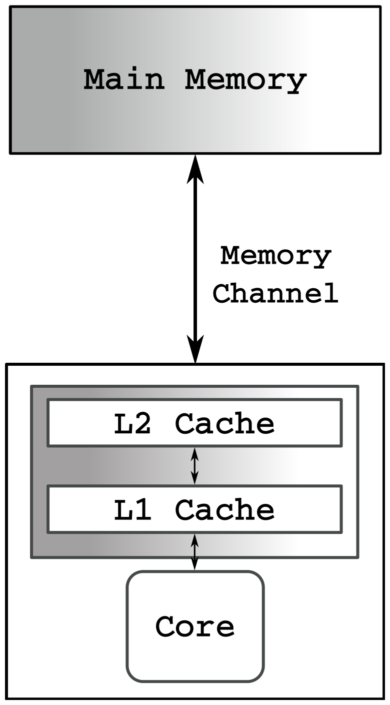
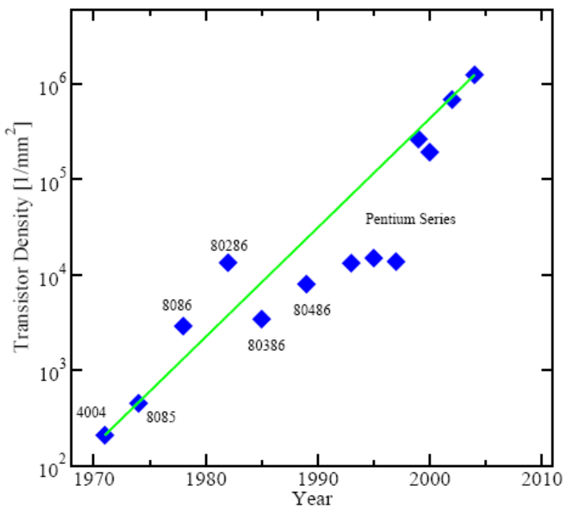


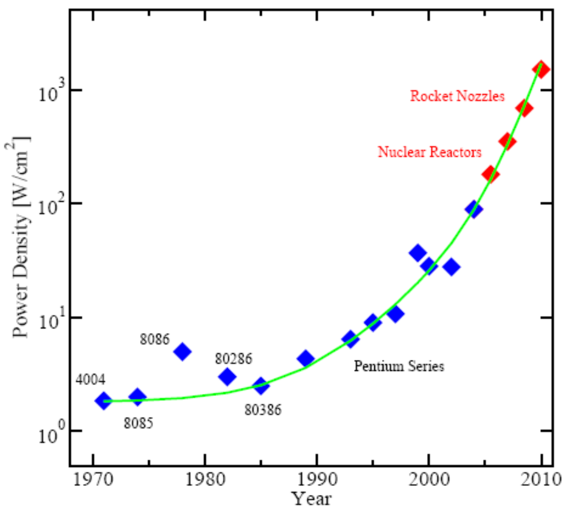
Figure 1.3: Single-core chip with memory

maximizing performance by increasing cache locality in the hierarchy for a specific application requires an enormous programming effort – to the point that humans typically achieve a tiny fraction of the theoretical performance [9]. Moreover, traditional cache hierarchies exchange memory bandwidth to improve memory latency. This is acceptable for sequential processing but is extremely inappropriate for the bandwidth-starved [10] highly parallel architectures found in field-programmable architectures. The negative impact of this technology has been documented and caused some to propose redesigning the standard OS/memory/network interface [10].

Up until 2007, the trend in computing was always to use the fastest core. The main reason for this is that Moore's law continues and has enabled conventional multi-processor scaling because of Dennard scaling. In 2007, the industry broke this trend of single core with the advent of multi-core chips that increased the parallelism but each core was slower than the contemporary single-core processor [11]. There are two main reasons for the end of conventional CPU scaling. The first reason is



(a) Transistor integration density per die



(a) Power loss density per die

Figure 1.4: Comparison of transistor density and power loss density over decades of microprocessors

that simply adding additional cores does not guarantee better performance [12]. The increase in the thermal density and power consumption of multi-core chips constrains the clock speed which then affects performance. Thus, power has also become a first-class design parameter. Figure 1.4 shows a comparison of the increase in transistor integration density per die and the power loss density in those chips. The transistor density grows exponentially as predicted by Moore's law, but the power loss density also increases exponentially on a logarithmic scale [13].

In an era of million LUT FPGAs devices, it is possible to imagine single chip, parallel architectures with thousands of computational cores. However, the number of I/O pins available to interface to off-chip memory remains extremely limited. Figure 1.5 illustrates this by showing multiple parallel cores and four channels to memory. The main solution to this was to use caches. This approach however, has a several disadvantages.

One of the most important disadvantages is the energy expended in data movement. With the use of conventional caches, the physical locality of the memory during

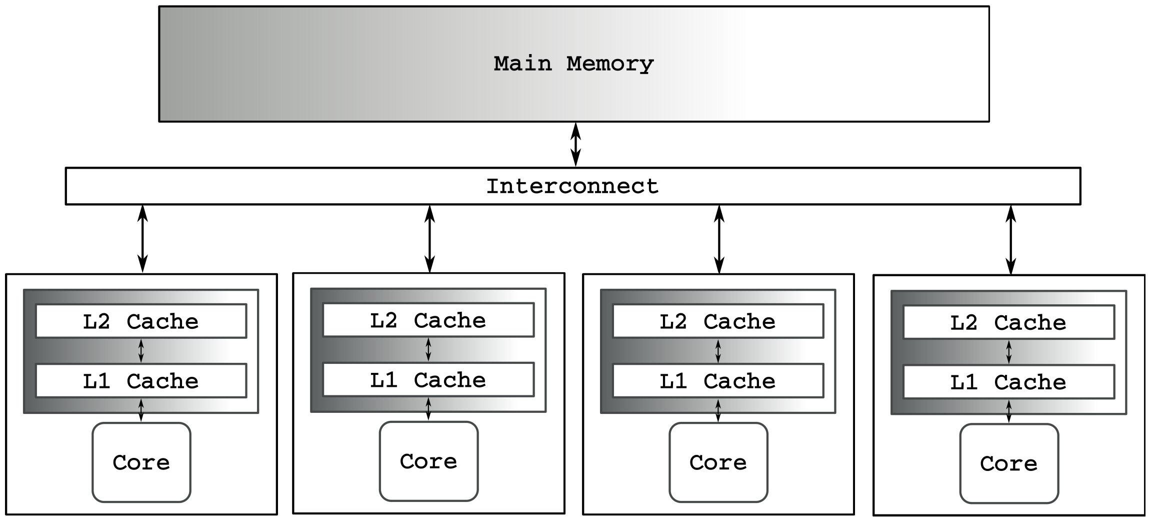


Figure 1.5: Multi-core chips with shared memory

a cache miss can result in a large amount of energy spent in moving the data. Also, the type of conventional cache (set-associative, direct-mapped etc.) entails extra circuitry to support the caching algorithms, without offering any "real" advantages. The energy burned is the same, and so is the unit of transfer (cache line). Another disadvantage with conventional caches is that the atomic unit of transfer is only a cache line which is typically 64 bytes or 128 bytes. This means that every time there is a request for data the probability of having to fetch a new cache line is higher and thus more energy is spent.

Although the energy spent and the atomic transfer size are disadvantages, memory bandwidth is a more significant issue in modern parallel system. For highly parallel architectures with many independent cores competing for the same bandwidth to off-chip memory, the traditional cache hierarchy approach is ineffective at reducing latency and wasteful in terms of potential bandwidth; this is an untenable trade-off for bandwidth starved parallel architectures. The use of caches definitely squanders bandwidth in favor of the illusion of lower latency. It devalues memory bandwidth in favor of statistically improving the memory latency observed by the processor. For a single core microprocessor, every cache hit is a success. However, for a memory-bound many core microprocessor, the goal is to fully utilize the off-chip memory bandwidth. As the cache hit rate increases, it leaves a reactive cache hierarchy guessing how to best use the available bandwidth, which sometimes results in doing nothing. Every unused cycle to off-chip memory is lost and cannot be recovered in the future. For these architectures, performance is often memory bound rather than compute-bound.

It is expected that future multi-core chips will exhibit characteristics that further exacerbate these issues. For example, there will be many more cores per chip than memory channels per chip, which increases the demand per channel. Likewise, it is unlikely that the chip will consist of a uniform set of cores. Rather, the chip will consist of some ratio of small, slow, energy-efficient cores for parallel processing and large, fast cores for sequential portions of the application [14].

There are emerging new technologies like phase change memory, 3D stacked memory and nanotubes. However, they have some shortcomings. Firstly, these new technologies are very expensive. Also, such technology is more target-specific and does not have much flexibility to be ported on different platforms with minimal effort. In addition to this, phase change memory has higher access latencies, and incurs higher power costs than conventional DRAM.

As mentioned, custom, fixed-function ICs have dealt with this growing inequity by introducing ever larger and ever deeper, reactive memory hierarchies. These conventional data caches are relatively simple and operate “in the moment” with no proactive, long term strategy. This is perfectly acceptable for single-core processors where temporal and spatial locality is high. But for multi-core processors, this approach will be ineffective. To address these limitations in the memory subsystem design, this work proposes an alternative architecture that introduces a new outlook on memory.

This dissertation introduces a proactive memory architecture that does not require specialized hardware and can be tightly integrated not only with the CPU but also with the network. This memory subsystem, when combined with an appropriate computational model, is designed to efficiently utilize the available memory bandwidth while hiding memory latency in highly parallel systems. The grand vision consists of a memory subsystem that includes the overlap of communication and computation, anticipates future communication, and reorders outstanding communication requests to keep memory channels fully engaged and utilized in the most efficient ways. The memory architecture also has the capability of handling variable-sized memory blocks that are larger than a cache line and hence can hold more data than a cache line, thus reducing the probability of going off chip to fetch data. The memory subsystem is called the *Active Memory Management Engine (AMME)*.

To accomplish this, three main elements are needed: an abstraction that separates the computational load from the memory subsystem via a set of banks of scratchpad memories per computational core, an interconnection network, and the proactive memory management subsystem. This imposes a few constraints on the computational load. First, it assumes that the application has a large degree of parallelism and that many independent tasks are available. These tasks are assumed to be relatively short, atomic sequences — small bits of imperative programs or finite state machines that deterministically progress from a start state to end state. The second assumption is that a computational core has the ability to be oversubscribed. That is, a core can queue multiple units of work and process them in the order that the memory subsystem provides them. Often, the second constraint can be addressed with MUXed banks of scratchpad memory per core.

Given these assumptions, the system level view becomes a stochastic process with computational cores providing a demand on the memory subsystem that can be represented by a random variable (an arrival rate) and the memory subsystem responding based on a random variable (a completion rate). In such a circumstance, the overall goal of the system is to balance computation against the physical limitations of off-chip memory bandwidth to create a queue of enough outstanding requests that the memory subsystem can intelligently order the transactions. This concept is known as Little's Law [15], which is expressed as

$$N = \lambda \times W$$

where N is the number of outstanding requests, λ is the arrival rate, and W is the wait time. In this case, λ, the arrival rate is the number of tasks created per second and W is how big the tasks are. Both can be manipulated by a suitable run-time system so that N does seizes (in theory) the on-chip resources. Hence, an artificial stationary process is created that allows the system to approach 100% bandwidth utilization as the on-chip resources increase. Combined with the latency-hiding technique introduced previously, this approach has the potential of dramatically improving the overall throughput of highly parallel applications on many-core architectures.

1.1 Thesis Statement

Many successful reconfigurable computing designs use streaming I/O to move data from source to sink through a network of computational kernels. In some applications the source or sink might be an instrument or storage (volatile or non-volatile memory). However, there are far fewer successful designs that include a large number of parallel cores, concurrently making random accesses to main memory. This is because every core observes the full cost of memory latency and, if the unit of work is too small, the tasks sequentialize on the single channel to main memory, i.e. they become memory-bound. The design presented in this dissertation aims to address the latter class of applications. Thus, the hypothesis is:

The use of an active memory subsystem with a flat hierarchy that is optimized for bandwidth, will support a larger degree of parallelism and ultimately reduce the time to completion for future many-core architectures.

To answer this question, we briefly present key concepts of the new architecture AMME, that are crucial to proving this hypothesis.

- Flatter memory architecture: The idea with this architecture is to move away from traditional cache hierarchies by developing a flatter hardware data-structure (B-tree) based memory architecture and local scratchpad memories per core.
- Programmer specified locality: Global byte-addressable global address space is a bottleneck for a multi-core system. Using named memory segments fair well in multi-core/many core systems. This research introduces a named memory segment scheme (SHA-like) instead of traditional global address space.
- Active memory system: Another disadvantage to existing caches is their tendency to be "passive". In this work, we develop an active system that can keep track of memory segments that do not have data associated with them yet. Furthermore, this active memory system also has the capability of promoting core-to-core communication without going off-chip.
- Localized restart: This feature enables adding a policy to enable *checkpointing*. This architecture uses an *epoch* mechanism to enable the localized restart of processor cores. This is also useful when individual processors need to recover from failures and go to a known state.

The implementation of these features is explained in more detail in Chapter 3. In this work we propose to include these design features to answer a set of questions through a systematic design analysis that includes the use of the new memory architecture in concert with traditional parallel applications that are memory bound and a new programming model called PyDac [16].

1.2 Metrics

The architecture described in this dissertation takes a new approach to memory that enables increasing bandwidth utilization and lowering memory latency while maintaining parallelism. We define a set of metrics that will help determine the impact of an active memory system.

Resource Utilization

Traditional caches utilize majority of the on-chip resources leaving behind a tiny fraction for the compute cores of the chip. Even with the high utilization of the on-chip resources, the deep hierarchy does not always facilitate faster access times. The AMME presents a B-tree based memory system with a flatter hierarchy. In addition to this, the resource utilization is measured for the hardware data structure. By exploiting the flexibility of this architecture, it is expected that the resource utilization will be much less that a conventional memory subsystem. Around 70% to 80% resources on chip are used for a conventional memory subsystem with caches. The idea is to curb the resource utilization to around 30% to 40% of the on-chip resources.

Access times

Since the implementation consists of a B-tree based hardware data structure, the access time to a node an important metric. Theoretically, the search (access) time for a B-tree lookup is $O(logn)$ [17]. Different sizes and data sets are implemented to test the B-tree structure in hardware and the access times in hardware are analyzed to see if a comparable lookup time is achieved. With the analysis of the first two metrics, we can answer the following question.

Named Memory Segments

This work introduces programmer specified locality through named memory segments. If there is no locality, then counter that with the amount of parallelism that can be supported. Using this scheme of named memory segments enables the processor cores to concentrate on the task of computation without having to keep track of the memory and thereby reducing overhead. In traditional byte-addressable memory, any interaction with memory for two (or more) processors includes an inferior mailbox system where the processors have to contend for the memory segment. With the introduction of write-once-read-only named memory segments, this issue is eliminated. Additionally, this architecture also structurally eliminates two of the three data hazards (Write After Read and Write After Write) [18]. This architecture enables large data transfers and offers explicit synchronization. And the naming scheme forces the locality to come from the programmer as opposed to conventional caches that reactively guess the locality and many times pay the penalty of unnecessary data movement. Naming schemes similar to this are mostly seen in imperative functional programming languages. One such example is Linda [19]. The biggest flaw in Linda was that since it is implemented as a software structure, it is not scalable.

Bandwidth Utilization

We conjecture that the full utilization of bandwidth to the memory device is increasingly more important than lowering the latency of memory systems. The B-tree based memory architecture proposed in this document will have multiple channels to the memory. With any memory subsystem we have a theoretical maximum bandwidth that utilizes the bandwidth toward useful computations. The memory subsystem presented in this document is an *active* memory subsystem. It is active because of its ability to begin transactions even before the data has arrived. That is, if a core is requesting a particular data by its name, and another core is currently in the process of computing the data corresponding to it, the memory subsystem keeps track of the name until the data arrives and then services the requesting core. This is explained in further detail in Chapter 3. The memory subsystem is also *active* in that it can make decisions of when to move data off-chip and how much data is to be moved off-chip. Eventually, we would also like to include the ability to reorder transactions within the queue of the memory subsystem based on the current demand. This can increase the effective bandwidth in terms of having the cores communicate with each other without going to off chip memory. Thus, the effective bandwidth also helps in determining the degree of parallelism that can be supported and also shorten the completion time. This metric will be calculated using real work load data from the green cores and also load generators with certain assumptions about the network. The goal is to analyze if we can prevent the queue from approaching infinity but keep the load at an “ideal” amount so as to avoid any core in the system to be idle and waiting for data.

Data Movement

Another major improvement over traditional caches would be for the AMME to curb unnecessary data movement. One of the biggest concerns with using the traditional memory architecture is that at any given time there is data that is being moved although that particular data may not be used. This leads to wastage of bandwidth. A traditional cache has a fixed cache line size that is always fetched. The core requesting the data may just need a fraction of the data that is being moved. Another angle to this metric is energy. A lot of energy required to move this data that is not efficiently used. In the new architecture proposed in this document, the memory subsystem can handle variable sized requests. This enables us to work efficiently and save more energy.

Average degree of parallelism (DOP)

For highly parallel systems, it has already been established that memory is the bottleneck to the system achieving peak performance. The focus of this dissertation is to dwell into the ability of a system to keep the parallel hardware operational at all times. Thus, we take into account the average degree of parallelism. The proposed architecture works actively to manage the requests coming in to increase the number of cores that are simultaneously operational and reduce processor core stalls. As opposed to a reactive memory system, the proposed architecture keeps track of requests that do not have data associated with them and issues a response when the data arrives. The average load on the processor cores will also dictate the average DOP achieved. Another factor to take into account is the number of memory channels. By increasing the number of green cores in the system and the number of memory channels the DOP can be computed.

Time to Completion

The goal of any many core system (as with the system presented in this thesis) is ultimately to reduce the time to completion for the task at hand. Although, it is intended in this dissertation to demonstrate a reduced time to completion, there are several outside factors that contribute to this metric. The Green/White architecture and the programming model (PyDac [16]) and the network [20] are being built be colleagues in the RCS Laboratory. The aim of this thesis is to use real workloads from [16] and also use a simulator to test the memory architecture. In the ideal case, all the green cores on chip will be active that will maximize the throughput leading to minimizing the time to completion. The issue of latency is addressed in the green cores instead of the memory subsystem by keeping the green cores oversubscribed.

The remainder of this dissertation is as follows. Chapter 2 presents the necessary background information the reader should be familiar with in order to understand the research presented, and covers related work. Chapter 3 covers the details and specifics of the design and implementation of this research. Chapter 3 lists all the implementations that led to the final design and structure of the AMME. Additionally, a stochastic model is presented that supports the motivation of this research. The next chapter of this dissertation is Chapter 4 which presents the evaluation and validation of this research. Lastly, in conclusion, a brief summary of this research and the overall impact of this work is discussed in Chapter 5. Also included in Chapter 5 is a brief section that presents possible areas for future research on this topic.

CHAPTER 2: BACKGROUND

This chapter provides a summary of the components and tools commonly used within computer architecture and reconfigurable computing. For readers who are familiar with these components it is advised to proceed onto Section 2.5; otherwise, it is recommended to review this material prior to continuing on with the remainder of the dissertation.

The next section provides an outline of Field-Programmable Gate Arrays. For a detailed overview of FPGAs, refer [21]. This will provide a better perspective of the components that can be found on modern FPGAs and are used in many of the designs discussed hereafter. For those more familiar with FPGAs reading this section may not be necessary, but it does provide a good overview and review for the less initiated. Section 2.2 briefly describes the types of memory devices. Jacob at al [22] provides an exhaustive description of the different types of memory devices and their working. Section 2.3 describes the basics of a B-tree data structure and the nuances of the structure used by this research. For further details on data structures, refer Cormen et al [23]. The chapter concludes with a related work section that describes the prominent research in this area.

2.1 Platform FPGAs

Platform FPGAs are Integrated Circuits whose functionality can be configured in the field. In other words, its functionality can be determined after it has been soldered into a product. They are System-On-Chip (SOC) solutions; they are completely contained within the FPGA[24]. An FPGA may be slower than an Application Specific Integrated Circuit (ASIC) but it is re-programmable, which is one of the most significant differences between the two. FPGAs consists of arrays of Configurable Logic

Blocks (CLB), I/O Blocks, routing networks and special purpose blocks. A high-level view of an FPGA can be seen in Figure 2.1. FPGAs contain hard IP and soft IP. Hard IP consist of components that are embedded into the FPGA fabric like CMOS transistors and soft IP which are flip flops or Look Up Tables (LUTs) that can be added or removed from the design. FPGA design is very flexible and thus is increasingly becoming popular in general purpose computing. FPGAs contain processors, on-chip memory, buses, bridges and networking and are thus capable of hosting entire systems. User application specific intellectual property can be designed and connected to the FPGA. Some available cores can be found in [25]. The study of such computing systems that incorporate user-programmable switches to determine functionality or architecture is called Reconfigurable Computing. The components that typically compose a SOC Platform FPGA and the tools that are used to synthesize designs on FPGAs are comprehensively explained in [21]. Figure 2.2 shows a Xilinx ML-605 developer board [26].

2.2 Types of Memory Devices

Memory is a crucial part of all embedded and reconfigurable computing systems today. In the case of small systems with not much need for storage, the designer uses small memory devices with fast accessibility. In this case, cost becomes a secondary issue. But larger systems will include different memory devices that have different access speeds, volatility and cost. The different memories presented in this chapter are Static Random Access Memory (SRAM), Dynamic Random Access Memory (DRAM), Block Random Access Memory (BRAM) that is found on FPGA devices. For additional details and to learn about other memory technology like disk storage (SATA) refer [22].

2.2.1 Static Random Access Memory (SRAM)

Although SRAM exhibits fast access times, it is expensive. Thus, it is used most commonly as RAM on-chip or cache memory in micro-controllers. It uses bistable

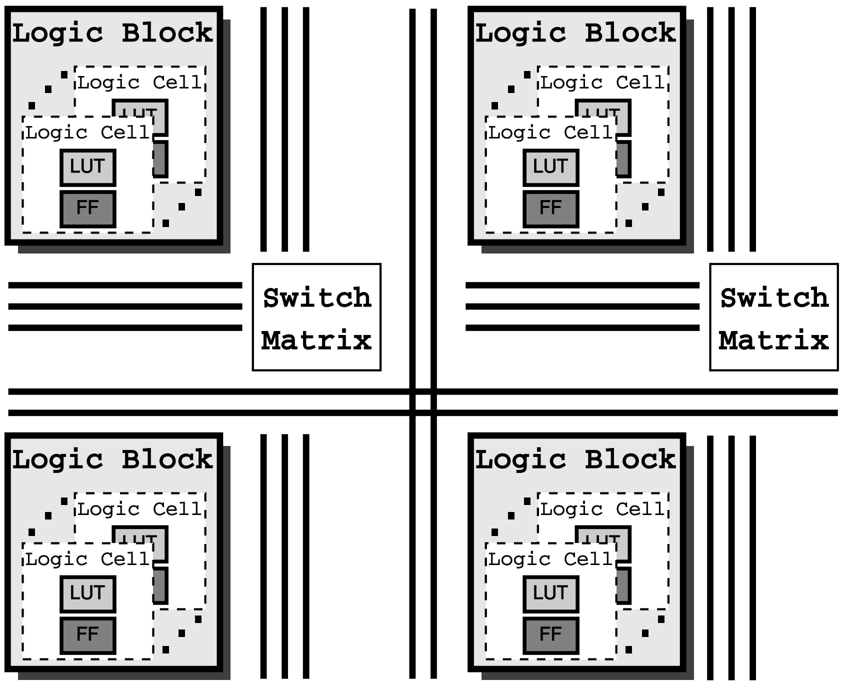


Figure 2.1: High level view of FPGA device

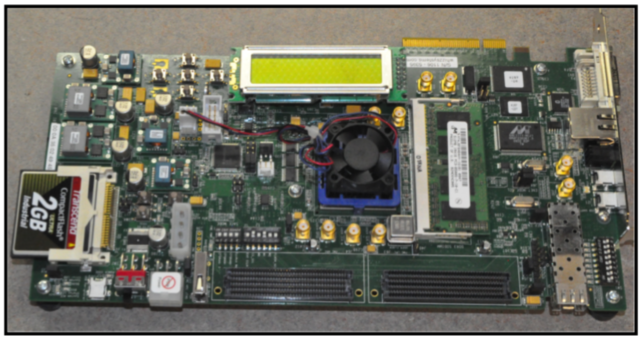


Figure 2.2: Xilinx Virtex 6 ML-604 developer board

latching circuitry to store bits of information.

2.2.2 Dynamic Random Access Memory (DRAM)

DRAM on the other hand, is less expensive and has slower access times and is most commonly used as main memory for personal computers. DRAM is made up of capacitors with an integrated circuit. Each bit of data is stored in the capacitor. Since capacitors leak charge, the information has a tendency to fade and need to be periodically recharged. Thus, it is called *dynamic* RAM. The circuit of a DRAM is less complex than the SRAM; it consists of only one transistor and a capacitor per bit of data stored, as opposed to SRAM that contains four to six transistors per bit of data. Synchronous DRAM is more popular as the DRAM is synchronized with the output bus. Also, the data storage is divided into various banks that allow a higher data access rate than asynchronous DRAM. The use of several banks facilitates several concurrent memory accesses. The interfaces have evolved over the years from SDR (Single Data Rate) DRAM, DDR (Double Data Rate) DRAM, DDR2 and DDR3 SDRAM. Although these interfaces show increased latencies, the minimum read or write transfer is 8 consecutive words in DDR3 SDRAM.

2.2.3 Block Random Access Memory (BRAM)

BRAMs are blocks of Random Access memory found in FPGA devices. The configuration and the number of BRAMs depends on the size of the FPGA device. Each block RAMs can be addressed through two ports but can also be configured as a single-port RAM. In addition, they are synchronous which means the outputs are registered. Many designs require the use of some amount of on-chip memory. Using logic cells it is possible to build variable sized memory elements; however, as the amount of memory needed increases, these resources are quickly consumed. The solution, to provide a fixed amount of on-chip memory embedded into the FPGA fabric called *Block RAM* (BRAM). The amount of memory depends on the device; for example, the Xilinx Virtex 5 XC5VFX130T (on the ML-510 development board) contains 298 36 Kb BRAMs, for a total storage capacity of 10,728 Kb. Local on-chip storage such as RAMs and ROMs or buffers can be constructed from BRAMs. BRAMs can be combined together to form larger (both in terms of data width and depth) BRAMs. BRAMs are also dual-ported, allowing for independent reads and writes from each port, including independent clocks. This is especially useful as a simple clock crossing device, allowing one component to produce (write) data at a different frequency as another component consuming (reading) the data.

2.3 Data Structures

A factor in large and complex systems is the storage of metadata that has to be accessed often and quickly. An effective method to store and retrieve the system's metadata would result in faster lookup times as compared to conventional memory controllers. This led to the investigation of data structures implemented in hardware to hold system metadata. Several factors have to be taken into consideration while implementing a data structure in hardware. Highly complex data structures that would be ideal in software, may not fare well in hardware because of the cost. This led us to first investigate the simplest data structure, a binary tree.

2.3.1 Binary Tree

A binary tree is a data structure in which each node can have upto 2 children. Nodes that have children are known as parents and the children are termed as left child and right child. The main node that serves as a starting point is called the root node. The search time of a binary tree is $O(n)$. The binary tree structure does not have the inherent capability to balance itself. This calls for additional administration to intermittently balance the binary tree.

Previously, a memory system using an ordered binary tree structure was implemented in hardware called the Dynamic Memory Allocation Controller [27]. The results definitely proved that it was feasible to implement a data structure in hardware but several issues rendered this structure not optimal for use in hardware. The

Table 2.1: Theoretical time complexity of binary trees in Asymptotic Notation

Operation	Average	Worst Case
Space	$O(n)$	$O(n)$
Search	$O(logn)$	$O(n)$
Insert	$O(logn)$	$O(n)$
Delete	$O(logn)$	$O(n)$

main issue with the binary tree structure was that a bad data pattern could lead to the tree growing in just one direction which would then become a glorified linked list. Also, the complexity of balancing the binary tree in hardware led to increased look up times, which proved to be unfavorable.

2.3.2 B-tree

A B-tree is a multi-way tree in which each node can hold multiple elements called *keys*. With more than just one key in a node, there can exist multiple children (branches) thereby containing the height of the tree. The B-tree is sorted by the *key*. There is other information that ties in with each *key* the B-trees. This is referred to as a metadata entry and is explained in detail in Section 2.4. The order of the B-tree (o) dictates the number of keys per node. For a B-tree with order o, the number of keys each node can have is $o - 1$. When there are more than $o - 1$ keys, the existing node is split to accommodate the new key. A B-tree, like a binary tree, grows downwards but facilitates re-balancing by splitting a node and adding it to its parent. Compared to the binary tree structure, the B-tree has several advantages. It can support more than just two children per node, which leads to reduced access times. Also the possibility of a bad data pattern leading to a linked list is reduced as the B-tree has a way of balancing out the data within a main branch. Additionally, B-trees are very popular in the field of Computer Science to keep account of data in an organized formant while allowing manipulations like searches, sequential accesses, insertions and deletions. They are commonly used in databases and filesystems because they are optimized to handle large datasets. Thus, the ideal next step to our investigation

Table 2.2: Theoretical time complexity of B-trees in Asymptotic notation

Operation	Average	Worst Case
Space	$O(n)$	$O(n)$
Search	$O(logn)$	$O(logn)$
Insert	$O(logn)$	$O(logn)$
Delete	$O(logn)$	$O(logn)$

was to implement a B-tree structure in hardware.

In this document, a variation of a B-tree structure implemented in hardware is presented. Knuth [28] defined B-trees in a particular way, but there are some discrepancies on the working and internal naming of the B-tree structure. This work assumes a B-tree that is derived from the original definition of the B-tree data structure with a few modifications to enable the effective use of the B-tree structure. To eliminate any confusion, and for the purpose of this dissertation, a set of assumptions that help define the structure of the B-tree are presented as follows:

- A B-tree of order (m) contains (m) keys in each node.
- The B-tree is sorted by the *key*.
- Every node can have at most m children.
- Every key in each node has useful metadata information including pointers that point to the location of the memory segment in hardware.
- All internal nodes are the same, including the leaf nodes.
- The root node functions exactly like all the other nodes.

An illustrated example of a B-tree indicating the nodes and keys is as shown in Figure 2.3.

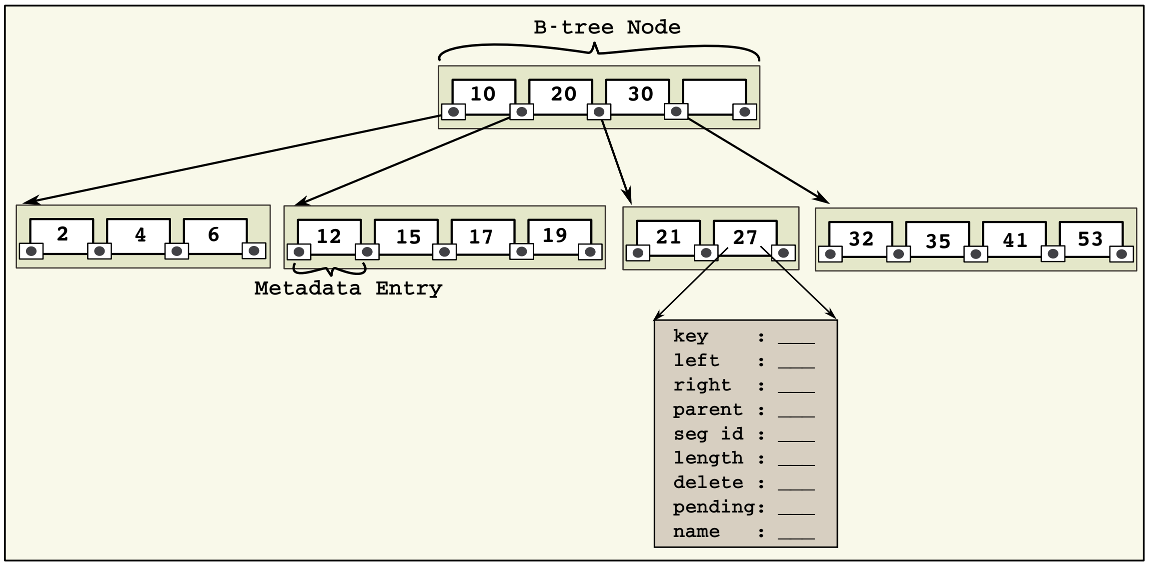


Figure 2.3: Example showing the relationship between nodes and keys in an order 4 B-tree

2.4 B-tree-Based Memory Subsystem

The plan is to use the B-tree structure to store the metadata, with node pointers to memory locations. The linked list serves as an LRU tracker, by keeping track of the least recently used data segments. A high-level illustration is as shown in Figure 2.4

The details of this implementation with a memory controller is explained in more detail in Chapter 3. However, some important definitions are presented here.

Order of a B-tree is defined as the number of children each node can have. If a node has k number of keys, the order of the B-tree will be $k + 1$.

Node of a B-tree is unit that can hold more than one key (as defined by the order) thus enabling the possibility of having multiple children.

Metadata Entry is all the information about a memory segment. There can exist multiple metadata entries per B-tree node.

Memory Segment is a sequence of bytes of data. This is stored in memory. This can be on-chip or off-chip memory.

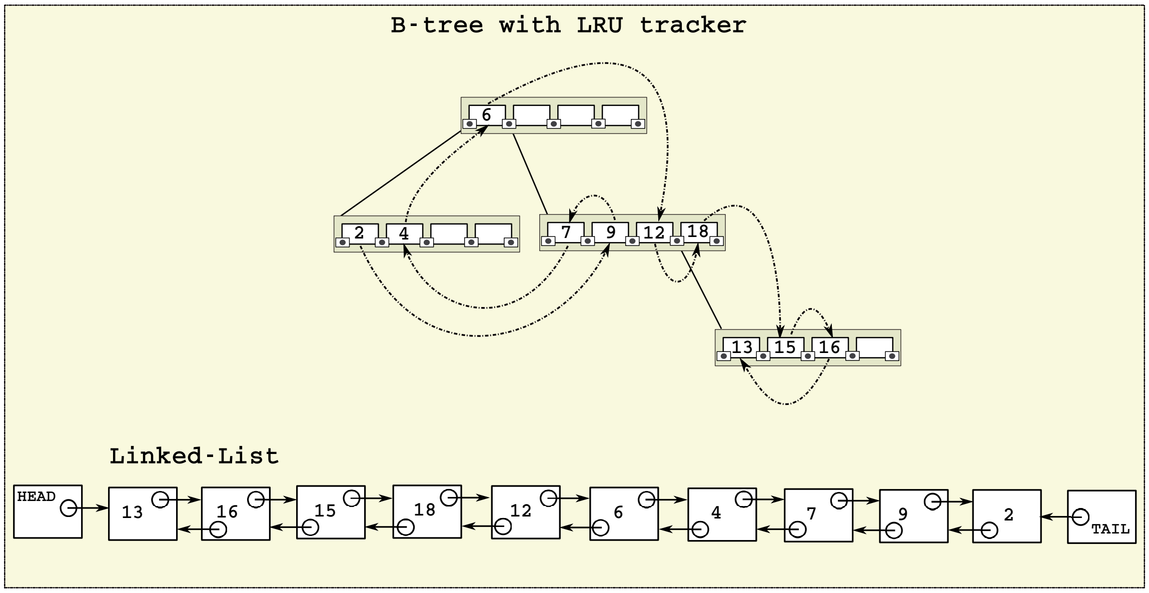


Figure 2.4: B-tree with a linked list

Segment Identifier is the direct location of the memory segment in the memory.

Key is the hashed name of a memory segment.

Name is the human-readable name of a memory segment.

Size is the size of the memory segment.

2.5 Related Work

The papers described in this section cannot be directly compared to this work but have an interesting approach to memory subsystems and share the same motivations as this dissertation. There are also papers that implement a data structure on an FPGA like [29], [30] and [31].

There is a plethora of research that has been conducted on caches and cache hierarchies. Most of this work focuses on how to mitigate latency issues at the expense of bandwidth with a very small percentage of papers focus on bandwidth. Smith [32] provides a comprehensive survey on the fundamentals of caches and their issues while also outlining the importance of memory bandwidth. Jacob et al [33] present a model that offers a solution to derive the optimal size of each level in the cache hierarchy.

Jacob's book [22], describes caches, DRAM, and disks in detail. Chow [34] and Agarwal [35] show that analytical modeling can be used to determine optimum capacity of a cache memory and close to accurate performance figures for caches. Smith [36] also presents how the cache line (block) size of a cache affects CPU performance. Denning [37] introduces the concept of "balancing" the memory and processor demands against equipments and present a working set model for memory management which is the smallest collection of information required to be present in memory in order to ensure efficient execution of a program.

The Fresh Breeze project [38] presents a parallel programming model for a multicore chip. The memory model of this project uses data objects as trees of fixed-sized memory chunks. The model uses global shared address space and write-once, read-only format. Although this model is also tree based, the main difference is that they implement the trees in software. Also, they work with fixed-sized memory chunks as opposed to the memory subsystem, AMME, described in this dissertation that can work with variable sized data chunks.

Liao, Zhu and Bhuyan [39] describe memory stalls as a major source of overheads in high speed network architectures. The authors identify page data structures to be a main factor in these memory stalls. Also, stated is that existing platform optimizations like Direct Cache Access are insufficient for network processing bottlenecks. They present a new server I/O architecture that shifts the DMA management to an on-chip network engine. Although the solution presented in this paper is focused on networking bottlenecks the motivation emphasizes the argument made in this dissertation. Saidi et al [40] show that the performance of systems is no longer dictated just by processor cores but in fact, the memory system, core interactions and I/O interactions play an important role in defining the overall performance.

Burger, Goodman, and Kägi [41] make a case for improving memory bandwidth over latency. The main point of this paper is that with the use of increased latency-tolerance techniques, memory bandwidth will the impediment to performance. In other words, bandwidth constraints will lead to higher latencies. The authors state that metrics such as average memory access time will not address the issue of memory stalls that are a result of insufficient memory bandwidth. The metrics they use include the rate at which external memory can supply operands and on-chip memory reuse. They also define a new metric, traffic inefficiency, that is the ratio of traffic generated by a cache and managed memory.

LEAP scratchpads [42] is a project by Adler et al that presents a new scratchpad architecture for reconfigurable logic that has the ability to dynamically allocate and manage multiple memory arrays that are independent in a large backing store. The use of LEAP scratchpads enables the automatic partitioning of on-chip memory. Another similar project is the CoRAM (Connected RAM) [43] that serves as a bridge between distributed computation kernels and memory interfaces. The CoRAM architecture provides a virtualized memory environment that can simplify application development and improve portability and scalability.

Lastly, with many new emerging memory technologies on the horizon, Kim [44] reviews PCM, STT-MRAM and RRAM and talks about their potential of becoming transformational memory technology. Swanson and Caulfield [45] talk about Non Volatile Memory (NVM) like Flash and Phase-Change Memory and the necessity to refactor, reduce and recycle to maximize performance with minimal disruption to the rest of the system.

CHAPTER 3: DESIGN AND IMPLEMENTATION

In the initial part of this chapter, an abstract model of the envisioned reconfigurable system is described, the stochastic parameters that can be used describe the behavior of our memory subsystem are defined, and the theory of operation presented. In the latter part of this chapter, the actual implementation of the design is presented.

3.1 Theoretical Formulation

With a high load on the memory subsystem, Little's law can be used to determine the specifics of the memory subsystem. The goal of the memory subsystem is to reduce the number of outstanding requests by servicing the requests at a faster rate thus reducing the wait time for each request. In such a system, all the factors are codependent. The number of scratchpads in the system can be used to determine the optimal the number of channels needed from the interconnect to the memory subsystem. This would dictate the load on the memory subsystem. In order to design and implement a system that would meet this criteria, it was essential to the feasibility of this idea. Toward that, Section3.1.1 presents a stochastic model that uses Little's law to study the possible advantages for designing such a system.

Additionally, the Green/White Architecture described in Chapter 2, is presented briefly in Section 3.1.2 as an abstract model from the point of view of the memory subsystem.

3.1.1 Stochastic Model

In this section, a stochastic view of the overall system is presented in tune with the memory subsystem. Some of the observations made here will be highlighted in Chapter 4 but first we define some common variables from queuing theory.

λ is defined as the average arrival rate of requests to the memory subsystem. For a stationary process, λ can also be called the average completion rate of a task.

n is the average number of outstanding requests in the system waiting to be serviced. This defines the load on the memory subsystem. These requests are stored in a FIFO as explained in Section 3.3.

t is defined as the average wait time for a request in the system. t can be represented as shown in Equation 3.1. The memory latency can be calculated depending on the type of memory channel the request uses. Ideally, the different memory channels can be Block RAM, SRAM, DRAM and SATA. The average wait time is the time the memory subsystem takes to service the request.

$$t = t_{memory\_latency} \times t_{avg\_wait\_time} \tag{3.1}$$

C is the total number of compute cores in the system.

B is the number of scratchpad memories per compute core. At a minimum, this number must be $2 \times C$ to facilitate the overlap of loading one bank with computation from the other.

ρ is the Utilization Factor. It is the expected arrival rate per mean service time, also called traffic intensity [46] or occupancy.

$$\rho = \frac{\lambda}{\mu} \tag{3.2}$$

As explained in Chapter 1, Little's law is crucial to the theoretical model and the computation of the variables in the memory subsystem. Little's law is stated as:

$$n = \lambda \times t \tag{3.3}$$

i.e. the average number of outstanding requests in the system can be determined by the average arrival rate of the requests and the average wait time. Assuming we have C number of cores in the system, each with B banks of scratchpad memory. The total number of compute cores in the system and the number of banks of scratchpads can be used to calculate the *maximum* number of outstanding requests the system can have as shown in Equation 3.4.

$$C \times B = max(n) \tag{3.4}$$

The service rate of each request μ can be determined by the wait time t. As explained in Equation 3.1, the time is given by the memory latency and the wait time to be serviced. We are currently working on a system with multiple links to the memory subsystem to enable concurrent accesses. The wait time in the memory subsystem is largely determined by the depth d of the data structure i.e. the number of objects in the data structure (in this case a B-tree). The service rate is as shown in Equation 3.5.

$$\mu \propto \log d \tag{3.5}$$

We are approaching this model stochastically by the application of the birth-death process. We also are assuming 10,000 tasks. This can be represented as shown in Figure 3.1. The arrival rate λ and the completion or service rate μ are random variables in the process. The main goal of this model is to ensure the full utilization of memory bandwidth across all channels to the memory. We have a programming model that can define the granularity of the system, thus the arrival rate can be dynamically tuned by the run-time system.

Suppose P_n is the steady-state probability of the number of outstanding requests in the system, we can say that

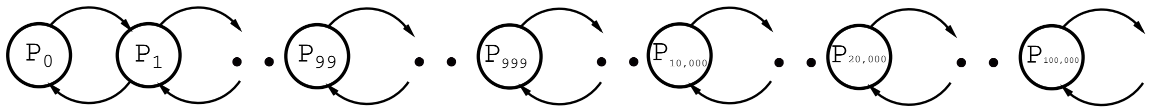


Figure 3.1: Birth-Death process

$$P_n = \frac{\lambda}{\mu}^n P_0 \tag{3.6}$$

where P_0 is the probability that the system has no requests and the queue is zero. If the system is in this state then there is wasted bandwidth which is not ideal. Using ρ and solving for the M/M/1 queue, we can derive,

$$P_0 = 1 - \rho \tag{3.7}$$

We also want the Utilization Factor to always be less than one which means that the arrival rate should always be *less* than the service rate. If ρ is one, the system approaches a state of chaos as the queue would continue to grow and when $\lambda = \mu$, the memory subsystem will be unable to handle the load and reduce the queue as the service rate will not be faster than the arrival rate. Thus, if we are in state P_0 for a memory channel, that means that we are not actively engaging the channel. We have exhausted the mathematical approach to solve this model and it is required that we work on the implementation in order to advance this abstract model.

Thus, the high level or system level view becomes a stochastic process with computational cores providing a statistical demand on the memory subsystem that can be represented by a random variable (an arrival rate) and the memory subsystem responding based on a random variable (a service rate). In such a circumstance, the overall goal of the system is to balance computation against the physical limitations of off-chip memory bandwidth to create a queue of enough outstanding requests that the memory subsystem can intelligently order the transactions. From a formal point of view, it is Little's law [15],

$$N = \lambda \times W$$

where N is the number of outstanding requests, λ is the arrival rate, and W is the wait time. In our case, λ (the computational load) can be manipulated so that N does not exceed the on-chip resources. Hence, we create an artificial stationary process that, as the on-chip resources increase, allows us to approach 100% bandwidth utilization. Combined with the latency-hiding technique introduced in the previous paragraph, this approach has the potential of dramatically improving the overall throughput of highly parallel applications on many-core architectures.

To realize the abstract model of computation presented in this section and utilize the stochastic characteristics, a key operation in the memory subsystem is ability dereference memory segment names. This operation takes a name and identifies the location of the memory segment. In the grand scheme, the segment might exist on chip, in DRAM, or even on a solid state drive.

3.1.2 Green/White Architecture

An high-level view of the Green/White architecture design is illustrated in Figure 3.2. As previously mentioned, this architecture consists of several cores, each with some amount of local memory (or scratchpad). There are two types of cores in the system — Green core and White core. The cores are differentiated based on how they are used. The White core is a modern, highly optimized processor core with a traditional cache hierarchy and intended for sequential parts of the application. The Green cores, which are the cores that use the novel memory subsystem proposed here, provide the parallelism. As such, they are either application-specific accelerators or processor cores that favor small size over speed (to increase the maximum degree of parallelism). In a new programming model and run-time system, described in [47], the White core is responsible for generating tasks that are sent to the Green cores for computation, assigning more than one task per core. Based on the tasks assigned, a

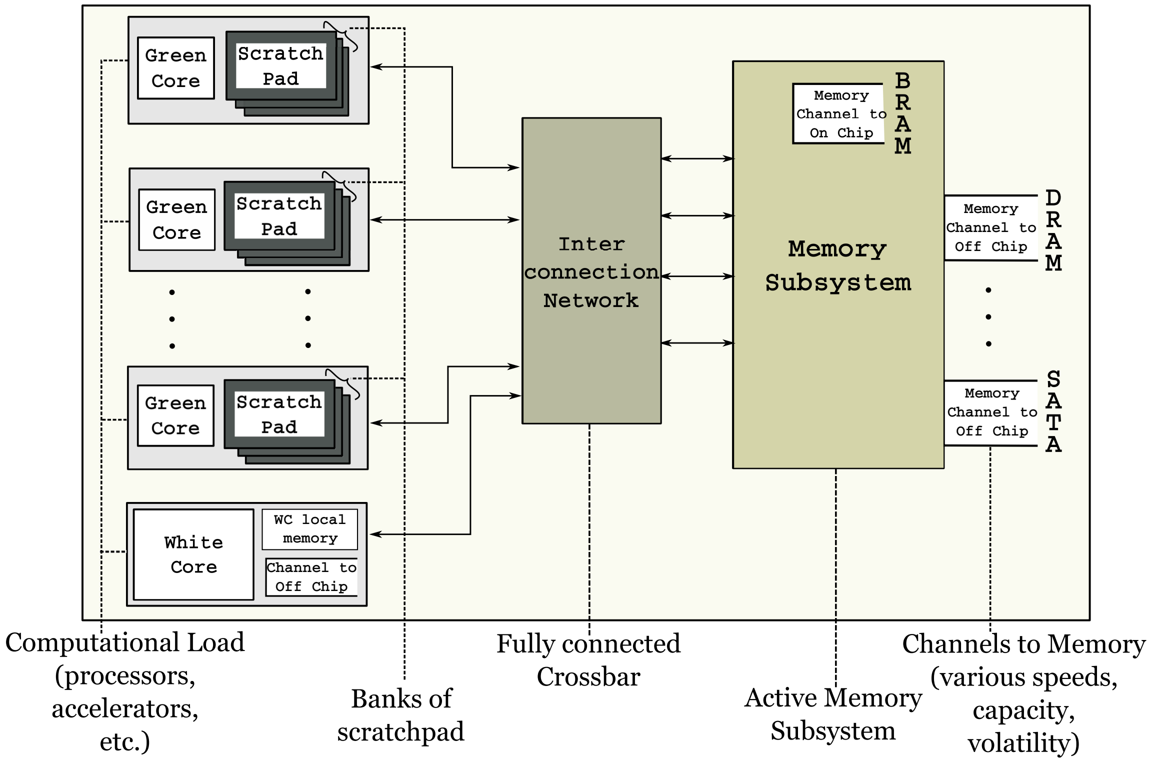


Figure 3.2: Abstract theoretical model

Green core will request the code and data *by name* for all of its tasks simultaneously. Once the memory subsystem has delivered all of the required data for a task to run, the task runs to completion and informs the memory subsystem that a new, named memory segment is available. The central idea is that that memory subsystem can be moving data to the scratchpad memories and from the scratchpad memories concurrent with tasks running on the Green cores. When a task completes, the selected scratchpad is changed and the core is reset. Note that under this scheme, there is no global address space, only named memory segments. These memory segments are locally read/write-able in an imperative style but once a task completes and the output shared with the memory subsystem, it is read-only. Communication and synchronization between the cores is handled by references to named segments. Names are ASCII strings that have been hashed to a fixed-width, binary digest, i.e. SHA-like.

This model does not allow for conventional synchronization primitives such as semaphores and it imposes a functional style of programming at the highest level ([47] describes the two-level programming model in detail). However, it does allow for state machines and imperative style programming at the task level and the organization overcomes several very difficult architectural challenges for extreme scale systems; one of which is the Memory Wall. Finally, as the illustration shows, there could be multiple channels to memory with different speed, capacity, and volatility characteristics.

3.2 Proof of Concept Implementation

The first implementation toward building a Memory subsystem for the system described in Section 3.1.2 was the implementation a core called the Dynamic Memory Allocation Controller (DMAC)[27]. This core consists of a binary tree data structure in hardware. The main aim with this implementation was to study the feasibility of implementing a data structure in hardware.

The DMAC consists of two specialized cores — *malloc* core and *free* core. The *malloc* core allocates blocks of memory from the DRAM and the *free* core deallocates the memory. A top-level DMAC controller core manages both the cores and hands off information from one to the other. The DMAC also consists of a Native Port Interface (NPI) that talks to the main memory. For the purpose of this paper, the DMAC is tested by using it as a slave on the system bus, but the end goal is to interface it to the local link network. Figure 3.3 shows the high level block diagram of the DMAC unit.

3.2.1 DMAC Controller

The DMAC controller is the central aspect of the DMAC core. It handles the incoming requests for data and issues the appropriate commands to the malloc and free cores.

The operands that serve as input to the DMAC are *command, size, dram_addr,* and *data*. *Command* encodes *malloc* and *free* requests. *Size* is the size of the memory

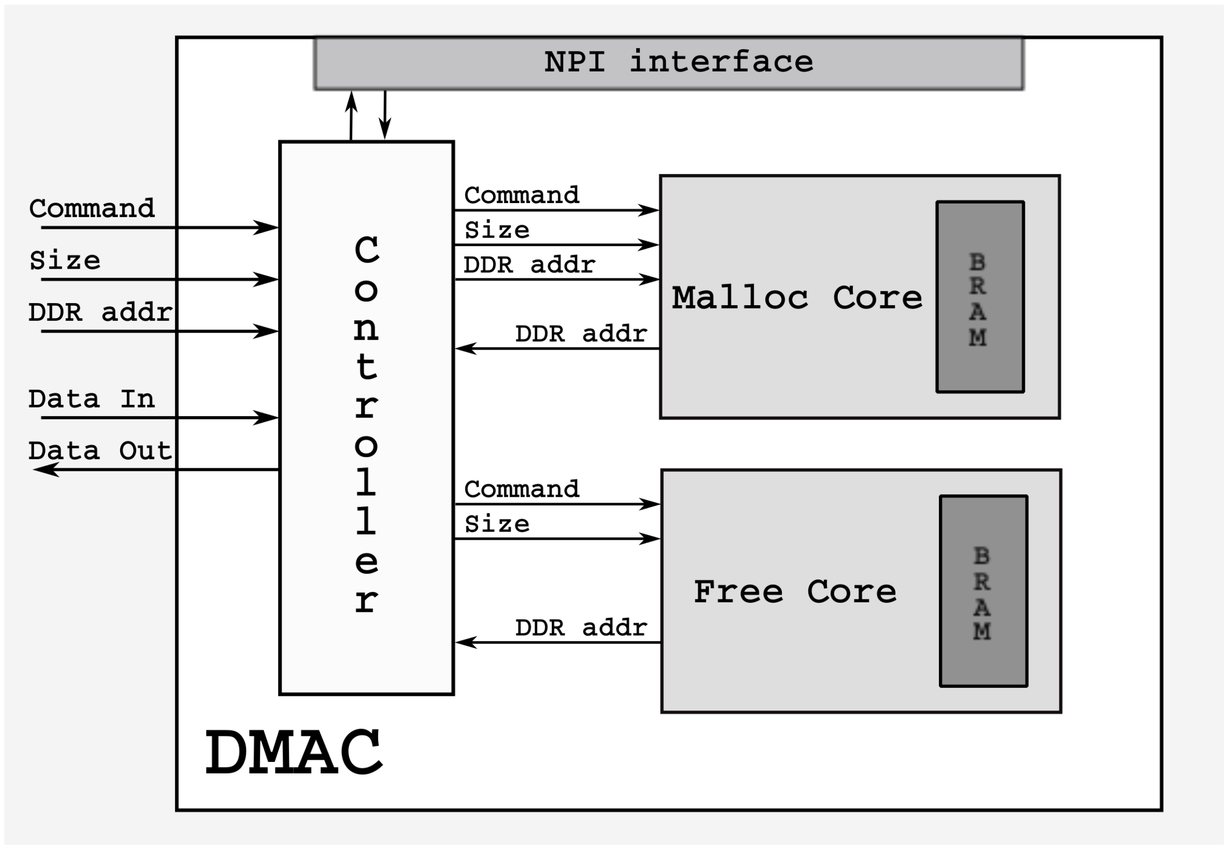


Figure 3.3: Block diagram of the DMAC

requested in bytes. And *dram_addr* is used to point to the block of memory to be deallocated.

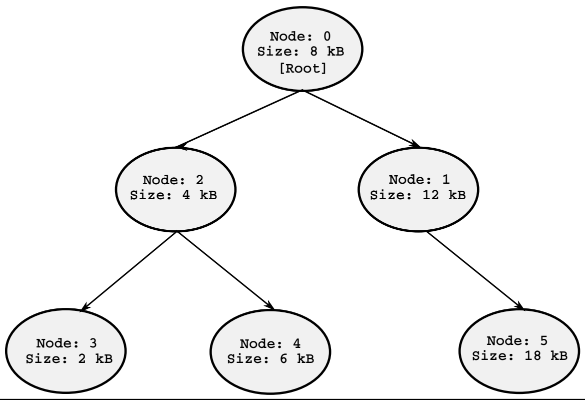
When a user issues a *malloc* command, the DMAC controller first looks in the free tree to find a chunk of memory that matches the requested size. The DMAC controller issues a *delete* command to the free core, if the size if found in the free tree, it returns a DRAM address to the DMAC controller. The DMAC controller then passes along this DRAM address to the malloc core and issues an *allocate* command. The DMAC controller keeps track of the latest available memory address of the DRAM in a register called *sbrk*. In the event that the size is not found in the free tree, the DMAC controller issues an *allocate* command to the malloc core and gives it the address from the *sbrk* register.

Similarly, when a user issues a *free* command, the DMAC controller issues a *delete* command to the malloc core which returns the DRAM address of the block that is freed, and the DMAC controller then hands that block to the free core with an *allocate* command. This way, the whole structure can be thought of as two lists — an allocated list and a free list of the memory.

3.2.2 Binary Tree Structure

The malloc and free cores are implemented as binary tree structures in hardware. The binary tree for the malloc core is implemented to sort by the DRAM address and the tree for the free core is implemented to sort by size. Each core has a list of allocated or free memory addresses stored in a table in a single port BRAM. The table in the BRAM consists of the following fields — size of the block, parent, left, right. Figure 3.4 gives an example of how the tree structure is stored in memory. This example is of a tree stored in the free core, hence it is sorted by size. Additionally, the DRAM address is also stored in the BRAM table.

The state diagram for the binary tree structures can be seen in Figure 3.5. When the malloc core receives an *allocate* command, it adds the node into the BRAM. If



BRAM address	Size	Parent	Left	Right
0	8	0	2	1
1	12	0	0	5
2	4	0	3	4
3	2	2	0	0
4	6	2	0	0
5	18	1	0	0
⋮	⋮	⋮	⋮	⋮

Figure 3.4: Example tree and corresponding table in the BRAM

the entered node is the root (the first node to be entered), the core waits for another command. The root node is loaded into a register (root register) to prevent the cost of fetching the root from the BRAM for every cycle. After the root node, for every *allocate*, the malloc core adds the new node into the BRAM, reads the previous node from the BRAM, updates the left or right field of the parent node (according to the *size* or *dram_addr*), and also updates the parent field of the new node that is entered into the BRAM. As mentioned before, the tree for the *malloc* core is sorted by the address of the DRAM (*dram_addr*). The set-up for the free tree is similar but the free tree is sorted by size of the memory block.

The malloc and free cores are set up to perform a *search* for a particular memory block. In the case that the core receives a find request, it requires an additional operand. This additional operand can be either the size of the memory block or the DRAM address of the first location of the memory block. To find a node, first the root register is checked to see if the node being searched is the root. If it is the root,

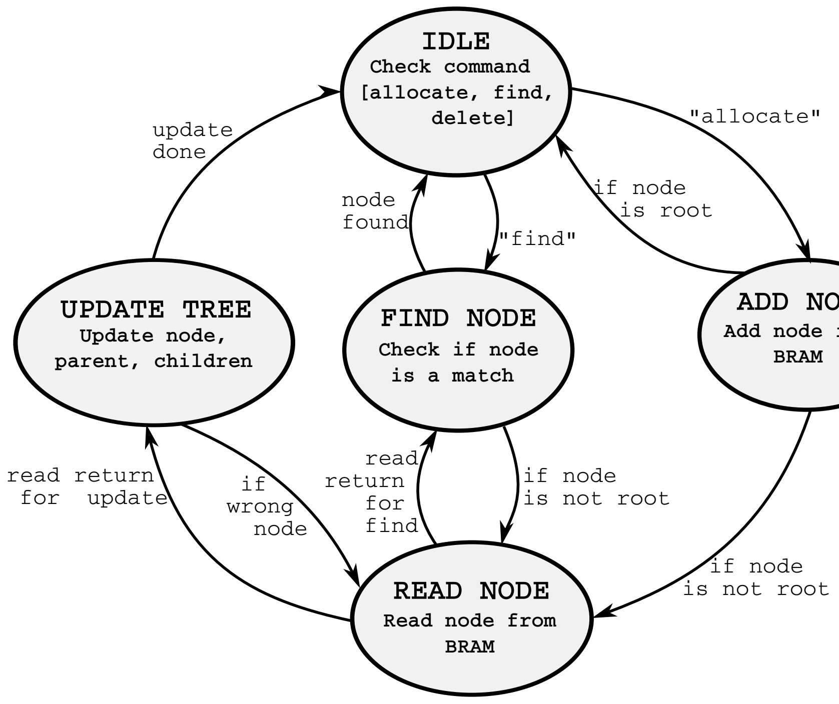


Figure 3.5: State Machine for the Allocate and Find Functionality of the Binary Tree Structure

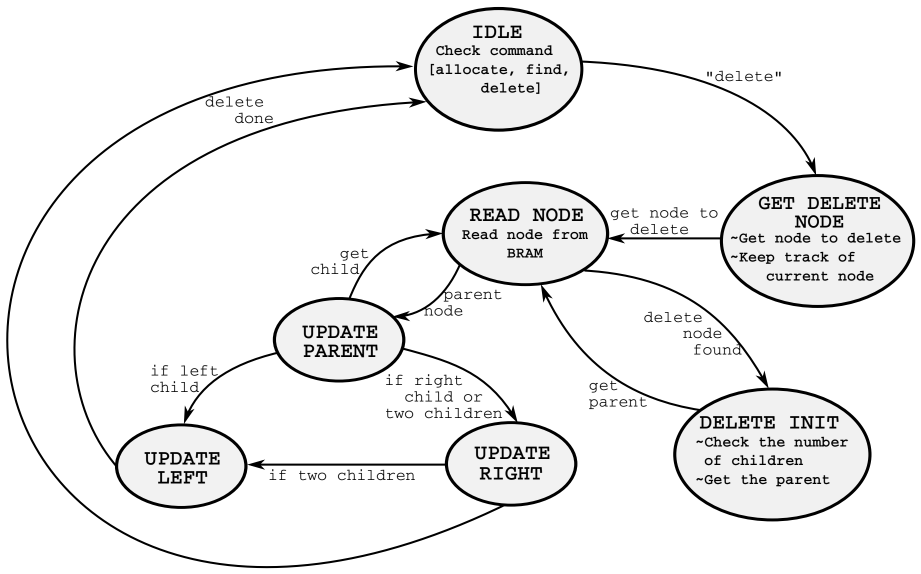


Figure 3.6: State Machine for the Delete Functionality of the Binary Tree Structure

it is handed over to the DMAC controller immediately. If not, the core traverses the tree to find the node requested. Once the node is found it is given to the DMAC controller.

When a *delete* command is issued to the malloc core, the start address of the block to be freed is given as an operand. The core first gets the node to be deleted from the BRAM. The delete function is complicated because the core has to update the child (left or right) node and the parent node. If the node to be deleted has two children, both the left node and the right node have to be updated in addition to the parent node. Figure 3.6 shows the flow of events when a *delete* is issued. Lastly, whenever a *delete* results in a “hole” in the table, the last entry of the table is swapped with the hole. Thus all the free nodes are at the bottom of the table and all the allocated nodes are at the top. This helps prevent fragmentation.

The previous implementation, definitely proved that it was feasible to implement a data structure in hardware but several issues rendered the Binary Tree data structure not optimal for use in hardware. The main issue with the binary tree structure was that a bad data pattern could lead to the tree growing in just one direction which would then become a glorified linked list. In addition to this, the complexity of balancing the binary tree in hardware led to increased look up times, which proved to be unfavorable. A desirable feature for the data structure would be to have less overhead of balancing, and hence the chief implementation of this dissertation consists of a self balancing B-tree data structure. The conceptual details of the working of the data structure itself, is presented in Chapter 2. The next section presents the chief implementation of the memory subsystem with a B-tree data structure.

3.3 Final Implementation

The Memory Subsystem hardware consists of a master controller, B-tree modules, and a BRAM pool for data. The aim is for this hardware to serve as an active memory controller, which not only services the requests that are coming in, but can also keep

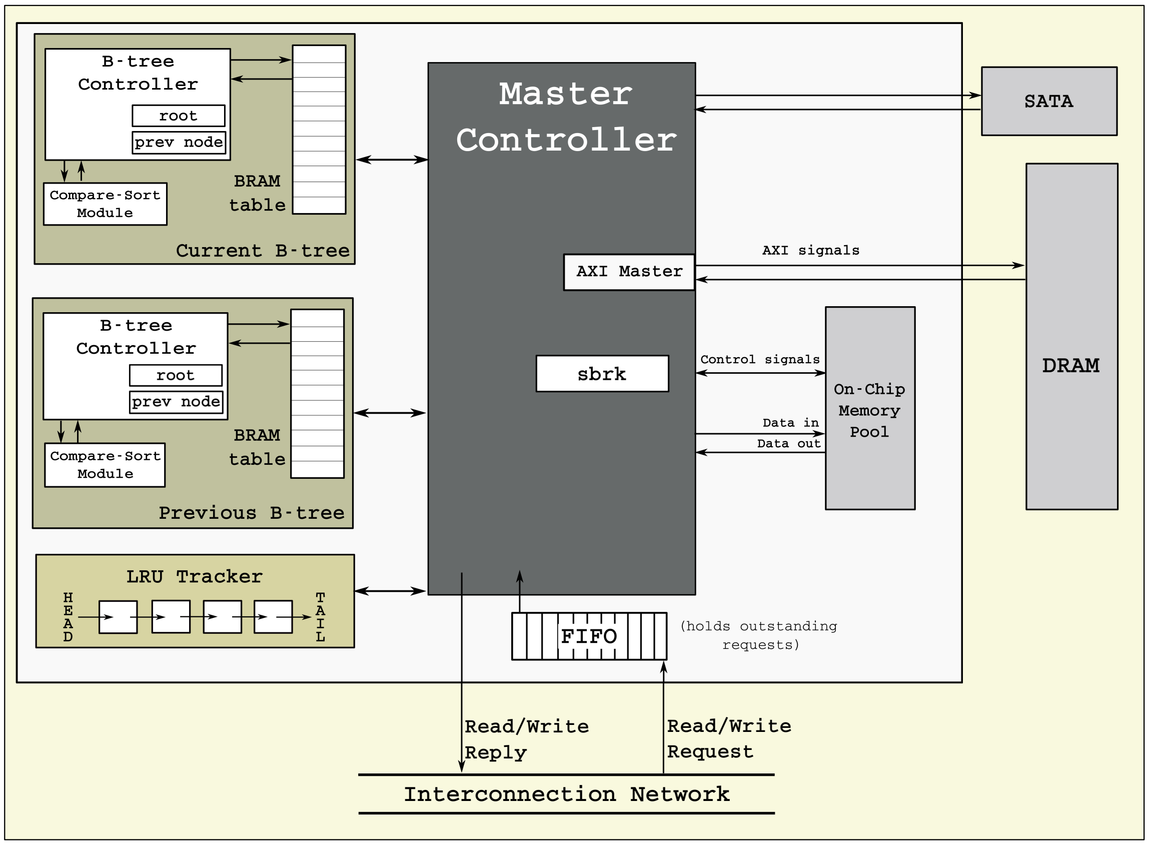


Figure 3.7: High level block diagram of the Memory Subsystem

track of pending requests. A unique feature of the Memory Subsystem is that it can work on variable sized memory chunks. Another feature is that it works on name based memory chunks that make it easy for the processor. Figure 3.7 shows a high level block diagram of the Memory Subsystem.

There are two versions of the implementation of this system in hardware. The first implementation was a B-tree Cache System and was modified to form the second version of the implementation Active Memory Management Engine (AMME). The core implementation of the B-tree data structure in hardware remains the same, but the second version has optimizations and better features. The next section describes both the implementations in detail, and explains the reasons for the differing implementations.

3.3.1 Name-to-Key Converter

The first piece of the implementation is the Name-to-Key converter, or the SHA-3 hashing core. This core is briefly explained in this section. Additional details can be found in [48]. The Name-to-Key Converter core uses human-readable, variable length names to identify the memory segments. However, this complicates the hardware which is far more efficient if the look-up key is a fixed number of bits. One of the features of this algorithm is that maintains its distribution property for different sized outputs. To use this in our experiments we created a Name-to-Key Converter core that accepts as input an ASCII name in human-readable form and produces a binary, fixed-width key as output. The key is suitable for tracking the memory segment within the memory subsystem. Figure 3.8 shows a high level block diagram of the Named-DMA core.

One hash characteristic to note is that a name does not always map to a unique key (i.e., it is not an injection). This means that we cannot recover the name for a key and there is the potential that two names might map to the same key (called collision when the hash is used in a data structure). In both cases, we address this with the small run-time system but the goal is keep this a rare event.

Creators of the Keccak hash function provide three open source implementations for FPGAs and ASICs coded in VHDL [49]. They state that there is a trade-off between area and speed given the symmetry and simplicity of its round function. Therefore, two implementations cover the two ends of the spectrum, a high speed core and a low area one. The third is a mid-range core. The one used for this work is the high speed core for its ability to be used in a standalone fashion [49].

The core is made up of three main parts; the round function, the state and the I/O buffer. The I/O buffer decouples the core from the bus used on a typical SoC, as we will see in our implementation. During the absorbing stage, the computation works simultaneously as input is fed to the buffer. The throughput is limited by the

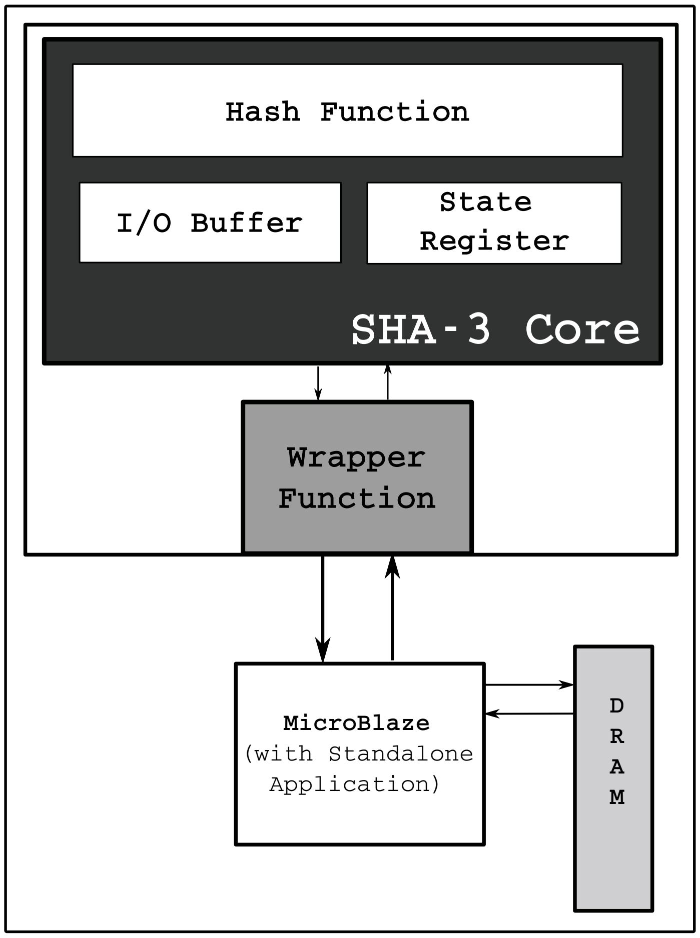


Figure 3.8: Block diagram of the Name-to-Key Converter core

width of the bus, which is typically 32 or 64-bit. For this implementation, 64-bit words are used as a default.

A customized wrapper served as an interface between the core's top level and the software controlled registers defined. Due to the 64-bit wide lanes in the internal permutation, the core is set up to input and output data over a 64-bit bus. Since the programmable soft processor MicroBlaze offers 32-bit word read and writes, two registers were written successively and concatenated in the top level of the hardware description files. After the second register with the upper half of the word is written, the I/O buffer feeding the core absorbs the first 64-bits of input for one clock cycle. Similarly, once the result is ready, the output is pushed into two registers to be read by the software application.

3.3.2 B-tree Cache System

The initial implementation of the B-tree memory engine consists of three main components: a B-tree table (BRAM table), a compare-sort module and a controller. This B-tree structure is used to hold the metadata of the system and is implemented in on-chip memory.

3.3.2.1 B-tree Memory Table

The first step toward setting up the structure in hardware was to implement a table in memory will serve as the B-tree. The B-tree memory table is a table residing in the BRAM contains the following fields — key, parent address, left address, right address, segment identifier (the base address of the data block in memory) and a couple of fields for checking validity and adjacency. Although the B-tree is stored as a table in hardware, in order to achieve the working of a B-tree, it is essential to keep track of the children (branches) and the parent. The size of the *key* is configurable for this data structure. For the purpose of storing the parent and left/right (branches) the BRAM address of the key which is the parent or left/right branch is captured. The node pointer field functions in two ways. This field is a 130 bits of which 4 bits function as control bits (*data_control*) that decide the content of the other 128 bits. The *data_control* bits can be set to be either local, BRAM pool or remote. These are the bits that dictate if the data will be stored on chip in BRAMs or off chip in the DRAM. If the actual data is four payload data words or less, the control bits can set to local and the payload data words are stored in the B-tree table itself. If the control bits are set to remote, the base address of the DRAM where the data will be stored is captured and the size of the data block is also stored.

3.3.2.2 B-tree Table Addressing

Each entry of the B-tree memory table stores the information for one *key*. Since each node of the B-tree contains no more than four keys, to perform any operation, at the very least, four keys will be required. Hence, the atomic unit of transfer between the controller and the B-tree memory table is a node (that will contain four keys). Thus for each added node, even if there exist empty keys, four contiguous memory locations are reserved. This saves the overhead of fetching the locations one by one. Any operation that warrants reading from the B-tree table, uses an address offset calculation to ensure reading one complete node as opposed to four contiguous locations that are part of separate nodes. For any given address, the corresponding read address is calculated by the formula:

$$read\_addr = addr - \quad \bmod \frac{(addr-1)}{(n-1)} \tag{3.8}$$

where read_addr is the address that the data is read from and always points to the base address of a node and n is the order of the B-tree. The logical view of the tree and how the nodes are stored in the B-tree table is as shown in Figure 3.9.

3.3.2.3 B-tree Controller

The B-tree controller unit is the top-level hardware that performs all the operations and manages the B-tree table. Keeping in mind the working of a B-tree, multiple consequent splits could result in a intermittent change of the root. The B-

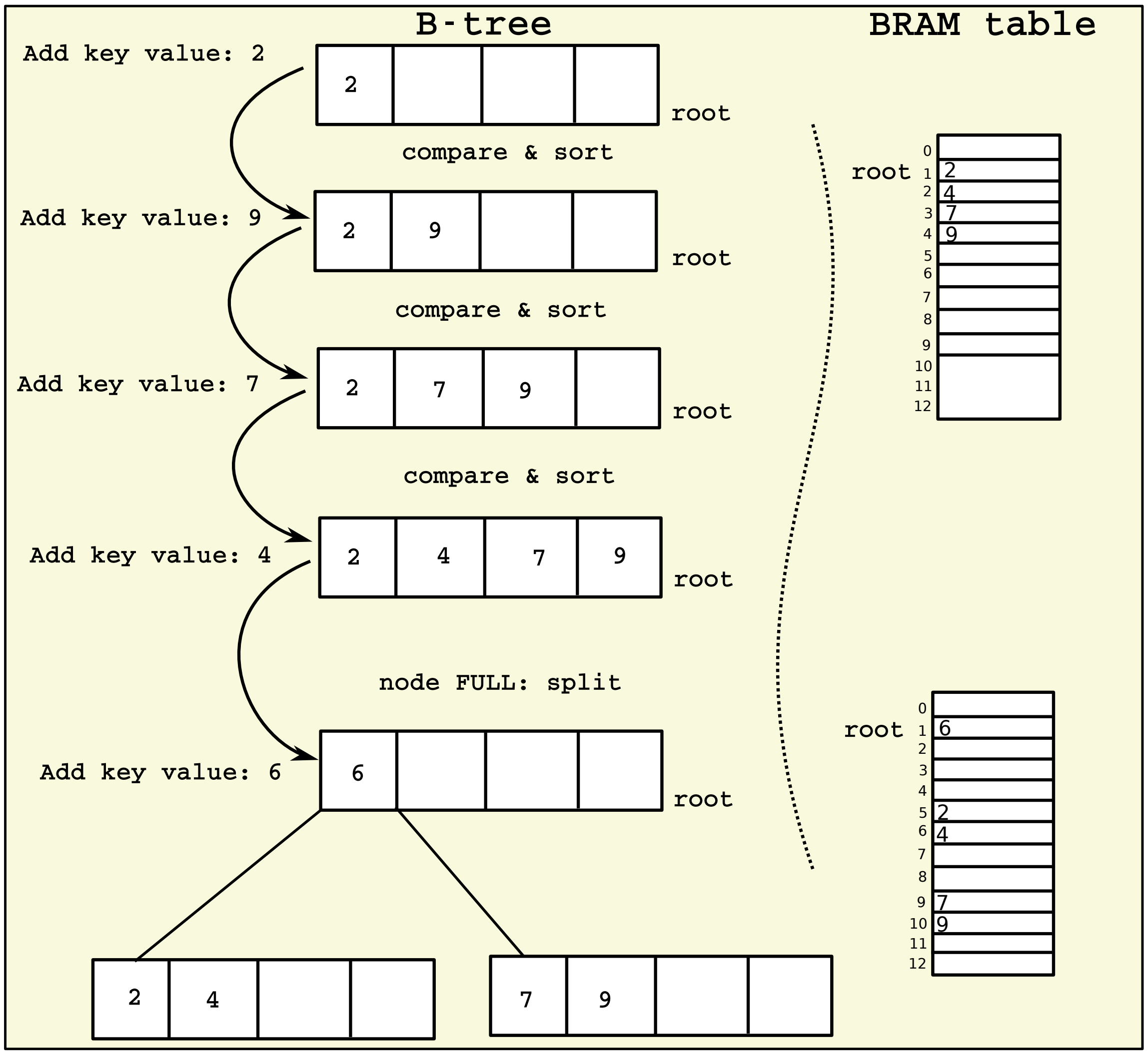


Figure 3.9: An example of a B-tree and the logical view of the nodes stored in the B-tree table

tree controller comprises of a root register that keeps track of the current root of the B-tree. The B-tree controller also has four *key* registers to hold the four keys of the current node in operation. Another feature incorporated into this design is that the controller retains the latest node that is fetched in a register on-chip. This results in faster look-ups of any keys near a recently referenced key. Every time a request comes in, this register is checked for a match before proceeding to the B-tree.

As in any conventional cache, the B-tree controller receives *Read* and *Write* commands. These commands are then interpreted by the B-tree controller as B-tree operations and processed accordingly. The B-tree operations include *add*, *find*, *split*, *update*, and *traverse*, . These commands are explained in detail below:

Add: On receiving a Write command, the B-tree controller adds the particular key to the B-tree table. In doing so, the B-tree controller first performs an add operation. Starting at the root, the new key is compared to the existing keys and added to the B-tree accordingly. If it is the first key to be added, it becomes the root and the root register is updated to reflect this key. If a root exists, the key is inserted at an appropriate empty slot in the tree.

Split: While performing an *add* operation, if the new key is to be inserted at a node that already contains four keys, a *split* is executed. A split can result in multiple splits, until the B-tree controller finds an appropriate empty slot to insert the new key.

Update: An *add* or a *spilt* operation warrants updating of parent/children addresses. A simple add operation would result in updating the parent to reflect the newly added node as its child. A split would result in a more complicated update operation, where the parent and the children have to be updated.

Traverse and Validate: An operation that walks the tree uses these operations. The traverse operation walks the tree, while the validate operation performs checks for parent/children.

Find On receiving a Read command, the B-tree controller issues a *find* to retrieve the requested key from the B-tree. This command typically traverses the tree looking for the said key. When the key is found, it retrieves the data packet associated with the key and passes it on the smart memory subsystem.

Delete or Mark for Delete The actual delete operation is performed on a higher level with epochs. On receiving a delete, the B-tree controller marks the key corresponding to the memory chunk to be invalid. Once the key has been marked for delete, that particular key is not considered while traversing the B-tree. It should be noted here that while the *mark for delete* operation renders the key to be invalid, it does not change the structure of the B-tree. Changing the structure of the B-tree is a very expensive operation and takes place on a *split* operation, and when an epoch is issued.

3.3.2.4 Compare-Sort Module

The keys in each B-tree node require to be sorted in order to achieve precise placement of the children. The input to this module is a node. This module then uses comparators to compare the keys and returns the node with the keys sorted in ascending order. Several operations of the B-tree require comparing and sorting the keys. The compare-sort module eases this operation, by making it a two-step, feed-in read-out process.

3.3.2.5 Master Controller

The master controller is the main pivot that controls the entire system. As shown in Figure 3.7, the master has control of two B-tree modules — the $BULK$ B-tree module and the $WAIT$ B-tree module. While the basic hardware of the B-tree modules is the same, the main difference is that the $WAIT$ module is used to store any request that comes in that does not have any data associated with it yet. Also, since there will never exist data associated with the requests in the *WAIT* B-tree, the B-tree table does not contain a field to store node pointers. Instead it makes a note of the core that is initiating this request.

For any *write* request that comes in, the master first checks in the *WAIT* tree if there has been a request for this data. If there has been a request, the master first services this request by sending the data associated with the request to the core as stored in the *WAIT* B-tree. Also, in the *WAIT* B-tree, this entry is now marked for delete. Additionally, this data is written into the *BULK* B-tree so any other read requests for this particular data chunk can be serviced accordingly.

The master controller stores the base addresses of the data blocks stored in the BRAMs in the *BULK* tree. Each request has a payload length associated with it. This controller keeps the BRAM pool in check and allocates and frees BRAMs from the BRAM pool. To ensure that the BRAMs are written to properly, the master controller uses a *sbrk* register. This register is the hardware equivalent of the memory management system call in Unix. Every time a new data chunk is stored in the BRAM pool, this register increments by the number of bytes equal to the payload length. Thus, this register always points to the latest available location in the BRAM pool. Additionally, there is a FIFO to queue multiple requests that are pending.

While the implementation of this design was functional, there were a few drawbacks that could be overcome. The B-tree Cache System, as the name suggests, has the ability to perform caching. As mentioned previously, the data_control bits are set to local, the payload data words are stored in the B-tree itself. When there is a request for this data, the B-tree traversals lead to the data itself reducing the need to perform additional lookups. While this is a favorable feature, this technique of storing the payload data words within the B-tree can be very restricting in terms of the size of the payload. As described in Chapter 1, a desirable feature for the memory subsystem is to have the ability to be *active*. But this B-tree Cache System behaves like a reactive cache, which led to the second version of the implementation, that is described in the next section. The implementation of the B-tree table addressing is also improved in the second version of the system. Furthermore, the *WAIT* and *BULK* tree are combined into one structure, decreasing the hardware footprint and making it more perceptive. The algorithm for this implementation is shown in Algorithm 1.

Algorithm 1 Master controller operation

```
if WRITE_REQUEST then
    find key in B-tree
    if found then
        Add data to the found key in the B-tree
        reply to the core that previously requested that data segment
    else
        Add key to the B-tree (with the data)
    end if
end if

if READ_REQUEST then
    find in B-tree
    if found then
        reply with data
    else
        Add key to the B-tree
        Set the pending flag corresponding to the key
    end if
end if
```

3.3.3 Active Memory Management Engine (AMME)

Many features of the B-tree Cache System were retained in the implementation of the AMME, with a few modifications. In this section, the modifications are described, with the assumption that the other features are maintained.

The block diagram of the AMME is shown in Figure 3.10. As seen in the block diagram, the B-tree module in isolation, is mostly the same.

Table 3.1: Fields in one metadata entry of the B-tree table

Field Name	Description
Key Value	SHA-3 Hashed Digest that is a unique identifier for the memory hunk
Parent Address	The B-tree table address of where the parent of the *key* resides
Left Address	The B-tree table address of where the left child of the *key* resides
Right Address	The B-tree table address of where the right child of the *key* resides
Adjacent Key	Indicates if a valid adjacent key is present
Size	Size of the payload
Pending Request	Indicates if the key has pending data and which core has previously requested the data
Payload Pointer	Contains information of where the payload data resides (which memory) and the base address for the payload
Valid	Indicates if the key is a valid key
Delete	Indicates if the key is marked for delete
B-tree Table Address	The B-tree table address for this particular key

3.3.3.1 Metadata Entry Fields

The modifications in the overall hardware of the system warranted some additional fields to the B-tree table. Table 3.1 shows all the fields in one metadata entry of the BRAM table with a brief description.

3.3.3.2 New B-tree table Addressing

One big change to the B-tree module (not depicted in the figure) is the addressing scheme. In the previous implementation, each BRAM entry held one *key* of the B-tree. The AMME has a wider BRAM table that can hold one *node*, i.e. all the keys contained in one node. This not only makes it easier for the B-tree Controller but it also enables faster access times. While previously it took four times the BRAM latency to fetch an entire node, it now takes just one lookup to fetch an entire node.

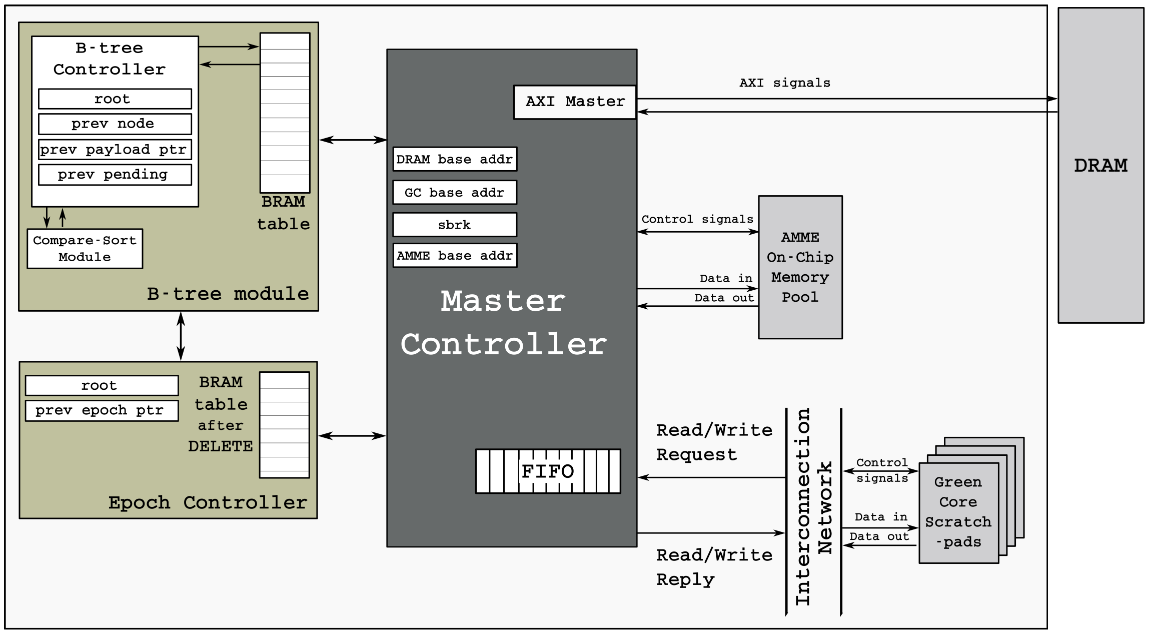


Figure 3.10: High level block diagram of the Active Memory Management Engine (AMME)

3.3.3.3 Epoch Controller

Another addition to the AMME is the epoch controller. This hardware facilitates the *delete* operation of the B-tree. When an epoch is issued, the Epoch Controller reads the B-tree table and actually deletes the metadata entries that were *marked for delete*. The Epoch Controller also conveys the deleted payload pointers to the Master Controller thus enabling their use in the current epoch.

3.3.3.4 AMME On-Chip Memory Pool

Previously, the data was stored inside the B-tree table. To eliminate the restriction of the payload size, an On-Chip Memory (BRAM) pool was introduced. While the B-tree module is used to store the metadata of the system, there is an additional resource on-chip to store the payload data. As the name suggests, it is a pool of multiple BRAMs that hold the data. The master controller reads from and writes to the On-chip Memory pool. The size of the BRAM pool is decided based on amount of resources available on-chip. This memory is used when all the scratchpads of a particular green core are full and the payload data needs to be moved out of the scratchpads to allow for newly computed data to be stored.

3.3.3.5 Channel to Green Core Scratch pads

The last modification that was made for the AMME was the addition of channels to the green core scratchpad memories. This makes it possible for the AMME to directly perform DMA on a memory segment residing in one green core's scratchpad to another green core's scratchpad. The Master Controller of the AMME has the base addresses of all the scratchpads of the green cores along with information on which scratchpad belongs to which green core. This feature enables a faster service rate for payload requests that are from a green core and the requested payload data resides in another green core's scratchpad.

CHAPTER 4: EVALUATION

The evaluation methodology of this research is split into different sets of experiments. The first set of experiments are centered around the Name-to-Key converter presented in Chapter 3. The second set of experiments include a detailed analysis of the B-tree hardware structure and supporting hardware. Next, the AMME unit is evaluated and the a set of experiments help examine the degree of parallelism enabled by the proposed memory subsystem and effective bandwidth achieved. Finally, an analytical study of energy expended by the memory subsystem is presented[1]. Each set of experiments and these will lead to inferences that attempt to prove the hypothesis presented in Chapter 1.

4.1 Name-to-Key Converter Results

For this investigation, all experiments were conducted on a Xilinx ML-605 development board with a Virtex-6 (XC6VLX240T) FPGA. Version 14.5 of the ISE/EDK tools were used to create and synthesize the design.

A base system running at a clock frequency of 100 MHz was built incorporating a MicroBlaze processor connected by an AXI bus to the SHA-3 peripheral core. 128 MB of DDR3 SDRAM memory was included in the base system to store large datasets of input for hashing. An RS-232 core operating at a baud rate of 9600 bps provides the ability to read the output of the SHA-3 core in a terminal via UART (Universal Asynchronous Receiver Transmitter). Additionally, axi_timer, a core that counts clock ticks, was included in the design to measure latency and throughput. XMD was used to download the bitstream onto the FPGA and the *elf* file to be run on the

[1]Although the intention was to integrate a non-volatile memory like phase change memory (Unfortunately, the RCS lab has not procured the funding to buy the infrastructure yet).

MicroBlaze. A text-based modem control and terminal emulation program, Minicom, was set up as a remote serial console and used to read out the hashes and write the output to a file so results can be analyzed later.

The hardware base system was tested for correctness by hashing the test vectors provided with the design and comparing the output of the simulation to those hashed on the hardware itself. The test vectors provided contain thousands of inputs and are in hexadecimal format. Inputs are 1024 bits long (16 64-bit words), whereas outputs are 256 bits (4 64-bit words).

After verifying the functionality of the SHA-3 core on hardware, the system was expanded to allow more robust experiments. The core was required to hash human-readable input of variable length. Hence, an algorithm was devised for that purpose. Additional details of the algorithm are published in [48].

Given that the size of the hash is a constraint (the size of the B-tree based memory subsystem [50] would be impractical otherwise) the output is truncated to different acceptable lengths when collecting data. Data with hashes of size 64, 32, and 16 bits were collected for a broad range study detailed later. Further precision experiments looked at median non-traditional lengths such as 20, 24, and 28 to optimize for lower area footprint.

4.1.1 Datasets

- Absolute paths to filenames on the RCS lab main server: The absolute paths to over 1,000,000,000 file names and directories were collected from a server, ensuring uniqueness. Hence, the data is not random, and is representative of real data where certain character patterns tend to repeat themselves. Random selections of different fixed sizes were taken from the main data collected for controlled experimentation into the effects of the number of names as it grows. The sizes chosen were 100,000, 50,000, 25,000 and 10,000 names. While large datasets were considered to study the scalability of our scheme, 10,000 in-fight

Table 4.1: Dataset statistics of Amazon fine foods reviews

Number of reviews	568,454
Number of users	256,059
Number of products	74,258
Users with > 50 reviews	260
Median number of words per review	56
Timespan	Oct 1999 - Oct 2012

transactions is a more realistic number to target. The reason for this is explained in Chapter 1.

- Amazon Fine Foods Reviews: The data set here is from the Stanford Network Analysis Platform (SNAP) [51] which is a general purpose network analysis and graph mining library. This library has a collection of about 50 large network datasets that include social networks, web graphs, road networks and more. The data set we propose to use is the Amazon Fine Foods reviews data set. This data set contains upto 500,000 reviews and the data spans a period of more than 10 years. The statistics of the dataset are as shown in Table 4.1.

  These reviews were made by real buyers of fine foods on Amazon's website. Besides being much larger in both, number of inputs and the input (message) size, the reviews offer a much larger degree of variability than absolute paths. The dataset cited contains 568,454 reviews, made by 256,059 users for 74,258 products. The median amount of words per review is 56, some constituting long paragraphs of product description. The format of the reviews is as shown in Figure 4.1

  The system had to be tweaked at the software level to absorb this size of message to hash. Preliminary tests conducted with 10,000 reviews showed *no collisions* at the 32-bit hash size. Again, this further supports our previous conclusions. However, more experiments should be done for larger input datasets and for varying sizes, due to the high variability in the input. Absolute paths contain

```
product/productId: B001E4KFG0
review/userId: A3SGXH7AUHU8GW
review/profileName: delmartian
review/helpfulness: 1/1
review/score: 5.0
review/time: 1303862400
review/summary: Good Quality Dog Food
review/text: I have bought several of the Vitality canned dog
food products and havefound them all to be of good quality.
The product looks more like a stew than a processed meat and
it smells better. My Labrador is finicky and she appreciates
this product better than most.
```

Figure 4.1: Example review on Amazon website

many repeatable prefixes such as forward slashes, as well as having almost identical portions in the path.

One setback observed from this test is the significantly slower performance due to the larger message length. The end goal is for the SHA-3 system to be implemented with the Green-White architecture, and such delays will not arise.

A number of post-processing scripts were written in order to extract useful information for the hashed digests. These scripts helped evaluate the number of collisions, time to first collision and the mean time between collisions. A detailed description of these scripts can be found in [48].

4.1.2 Performance

To assess the performance of the Name-to-Key Converter, the latency and throughput were measured by using the standard Xilinx axi_timer. The methodology used was straightforward. To measure latency, the timer is started right before the input is fed to the core. As soon as the core is ready to produce the output, the timer is stopped. The latency is the value that results from subtracting the first time stamp from the second, and is in clock ticks. Hence, the latency is the time between when the input is received and when the output is produced, from the processor's point of view.

The number of ticks recorded between the two events were 1139 clock ticks. At 10 ns per clock cycle, the latency is 11390 ns, or 0.01139 μs. Throughput was measured by feeding the core a known amount of names and measuring the wall clock time. Experimentally, the throughput was calculated to be approximately 38.65 Mbits/s at a 100 MHz clock rate.

As expected, the largest percentage of the time taken to compute hashes is spent in the processor. Read and write functions constitute the largest overhead, typically in the range of 30 clock cycles or more.

4.1.3 Size and Collisions Study

4.1.3.1 Broad Range Collisions Study

To study the effects of the size of the dataset on collisions at varying output lengths, a broad range experiment is conducted for datasets of 100,000, 50,000, 25,000 and 10,000 names. Three traditionally used sizes for the hash output are considered; 64, 32 and 16 bits. Hence, this experiment was run for each dataset (100,000, 50,000, 25,000 and 10,000), at different hash sizes (64, 32 and 16).

Collisions are counted by first hashing the entire set of names using the Name-to-Key Converter Core. After collecting the hashed output in a file (one hash per line), post-processing of the data can be done using a bash script. The script first sorts the hashes alphabetically, then uses the `uniq` command to count the duplicate lines and print the number of duplicates it found. It then sorts the lines in decreasing order, therefore providing data such as the total number of collisions, and the number of names that mapped to each individual hash. Hence, collisions can be observed closely to monitor for any possible trends or unexpected spikes in collisions at a certain point.

For 64-bit hashes, no collisions were observed at any dataset size. Only two collisions were reported when a 32-bit output was used to hash 100,000 names. However, no collisions were found when hashing the smaller datasets. The number of duplicate hashes for 16-bit hashes was high at approximately 30% for the dataset with 100,000 names, and around 7% for 10,000 names. While this is in line with expectations, it is important to validate the theoretical assumptions through an implementation on an FPGA. These results are presented in 4.2.

Table 4.2: Collisions observed with variable hash sizes

Dataset Size	Hash Size (in bits)	Number of Collisions
100,000	64	0
	32	2
	16	29,468
50,000	64	0
	32	0
	16	11,772
25,000	64	0
	32	0
	16	3,762
10,000	64	0
	32	0
	16	730

In terms of the AMME memory subsystem, the number of memory names in each set would represent the number of concurrent memory transactions (either pending or complete) in an epoch (between delete cycles). While the largest dataset used to conduct this experiment included 100,000 names, this is about 10× the number of in-flight memory transactions that was estimated to be needed in a future Exascale chip. These results suggest that a large hash size (64 bits) would not cause collisions, even as the number of memory transactions continues to scale beyond the 10,000 target. Furthermore, a hash size half as large (32 bits) would have relatively few collisions that the run-time system would be required to clean up. The number of collisions is too high for a practical system when the hash is 16 bits, even for 10,000 names. For that reason, it is deemed as an unacceptable size.

4.1.3.2 Hash Size Optimization Study

Initially. the AMME was based on the assumption that there would be 4096 in-flight transactions and a hash size of 16 bits [50]. The memory subsystem consumed roughly 11% of the chip resources. Given that caches typically occupy a much larger percentage of the chip's resources (up to 90% in some cases), this was considered a reasonable result.

However, the results presented in 4.2 suggest that this is not scalable to future, more highly-parallel designs. 16-bit hashes did not prove to be a viable option, even for the immediate target of 10,000 transactions. To get a better picture of how many bits a hash should be, a second experiment was conducted to hone in on the optimal hash size. The experiment was conducted similarly to the previous one. However, non-traditional bit widths such as 20, 24, and 28 were taken into consideration. The requirement is that the number of collisions remain low enough to be manageable, while keeping the total resource utilization on the chip reasonable.

Table 4.3 shows these results in detail.

Table 4.3: Collisions observed with untraditional hash sizes

Dataset Size	Hash Size (in bits)	Number of Collisions
100,000	28	16
	24	291
	20	4,565
50,000	28	5
	24	70
	20	1,200
25,000	28	0
	24	14
	20	279
10,000	28	0
	24	14
	20	279
10,000	28	0
	24	5
	20	54

For a sample of 10,000 and 25,000 names, 28-bit hashes appear to have no collisions, suggesting that this may be the optimal size for the target. Moreover, even with 50,000 memory transactions per epoch, only five collisions occurred. Considering that the run-time system would include a mechanism to resolve collisions, this low likelihood of their occurrence is reasonable.

A discussion of the cost-benefit analysis of choosing between 32 and 28-bit hashes presents itself here. While the results of this experiment suggest that 28 bits per hash provides enough security against collisions for the foreseeable future, one might ask if the additional area resources saved by opting the smaller hash is worth it when compared to that of the slightly larger 32-bit hash. A choice can be made here, depending on the resources available and how dire the need to decrease the utilization of the memory subsystem is. If the benefit is not substantial or required, then choosing hashes of 32 bits would be one way to over-engineer the memory subsystem for less collisions as designs continue to scale upwards.

4.1.3.3 Confidence Interval Study

A confidence interval is used in statistics to determine the reliability of an estimate. While the results presented in the aforementioned experiments are not estimates, but actual data produced by the hardware implementation, it remains necessary to determine whether the results are consistently repeatable.

For this purpose, five random seeds of each dataset size (100,000, 50,000 and 10,000) were collected for repeating the broad range experiment at hash sizes of 64, 32 and 16 bits. This means that the experiment was conducted 5 times, with a different set of names on each run. The results of the tests run for 16-bit hashes are presented in Table 4.4.

Table 4.4: Confidence interval study for 16-bit hash results

Dataset Size	Number of Collisions					
	1	2	3	4	5	σ
100,000	29,587	29,531	29,540	29,548	29,575	23.8
50,000	11,666	11,751	11,702	11,780	11,728	39.3
10,000	693	705	740	692	741	24.5

Since no collisions are ever observed for 64-bit hashes, the standard deviation of the 5 runs is 0. The standard deviation is also 0 for the experiments on 32-bit hashes at 10,000 and 50,000 names, but 0.836 at 100,000 names (where some negligible variation occurs). For the 16-bit hashes, the standard deviation is 24.5, 39.3, and 23.8 for 10,000, 50,000, and 100,000 names, respectively. The results show very similar numbers of collisions, as shown by the negligible standard deviation (with respect to the size of the datasets), proving that the number of collisions to expect is independent of the data, and the Name-to Key Converter should behave predictably with different sets of names.

4.1.3.4 Collisions and Keys

If the SHA-3 hashing function is to be used in a larger memory subsystem, it may be useful to know how many names can be hashed before the first collision. An experiment that analyzes the data to produce that metric provides an estimate for future consideration. This is called the Time to First Collision in this thesis. At 64-bit hashes, no collisions were reported. However, at 32-bit hashes, the first collision found with the dataset studied occurred after $71,303$ hashes. In other words, the two collisions witnessed take place well into the dataset name space. If the AMME is to target 10,000 transactions, designers can count on a safety margin given collisions do not occur early on for some reason. 16-bit hashes, on the other hand, offer no cushion. A first collision occurred after just 140 names. This further proves that 16-bits are not enough for producing reasonably unique memory segment names, and should not be used.

Another metric that can be of interest, is the Mean Time Between Collisions (MTBC). The idea of this metric is borrowed from the fault tolerance metric, Mean Time Between Failures, or MTBF. Since collisions are considered failures in this context, it is useful to know *how often* these failures occur. This is especially important when designing a system that deals with collisions, in that it provides an idea of how frequently that routine would be required to run, which may affect how long (in cycles). However, the MTBC is not a measure of time, but a measure of how many hashes (in-flight transactions) occur between collisions, on average.

As stated, experiments done with 64-bit hashes showed no collisions. However, the MTBC for 32-bit hashes is $49,883.5$. For 16-bit hashes, the MTBC is about half, $25,820.5$. These results strongly suggest that the SHA-3 hashing function distributes the hash values evenly across the available range.

4.1.4 Analysis of Using Hashes for Named DMA

Two aspects of these results are analyzed. In 4.1.4.1, the observed collisions are compared to the theoretical model that predicts hash collisions in a perfect hash function. In 4.1.4.2, the scalability of this approach is addressed by looking at how fast the number of collisions grow with increasing memory transactions.

4.1.4.1 Probability of a Hash Collision

An underlying assumption of hash functions is that the same input always produces the same hash, and if the function is a good one, a different hash is generated when given different inputs. However, collisions are unavoidable. Whenever a large set of names is mapped to a relatively small amount of bits, duplicates will inevitably occur. This is in line with the pigeonhole principle. Collisions are not always undesirable. When used in applications where fingerprint of DNA data is hashed, hash functions are designed to maximize the probability of collisions for distinct but similar data. In digital signatures and security applications, it is important that collisions do not occur, even under collision attacks.

Since this thesis studies the collisions of the SHA-3 hashing function in its use to generate named memory segments, it is important to see how the results compare to the probability of collisions from a mathematical point of view.

One way to look at this question is as follows: Given k randomly generated values (representing the number of hashes), where each value is a non-negative integer less than N, *what is the probability that at least two of them are equal?*

To answer the aforementioned question, it is best to find the probability that all values are unique, and subtract that value from 1.

Given N possible hashes, $(N = 2^n)$, where n is the number of bits that constitute the hash, assume a single value is mapped. Consequently, there are now $N-1$ remaining values that are different from the first. Hence, the probability of generating another unique value is $\frac{N-1}{N}$. Subsequently, there are $N-2$ remaining values that are different than the first two. Since each hash is an independent event, the probabilities can be multiplied to find the statistic for all N. In general, the probability of randomly generating k hashes that are all unique is:

$$\frac{N-1}{N} \times \frac{N-2}{N} \ldots \frac{N-(k-2)}{N} \times \frac{N-(k-1)}{N}$$

As this can be tedious to compute for large values of k, a safe approximation derived by using the Taylor expansion [52] of e^x is:

$$\mathrm{e}^{\frac{(-k)(k-1)}{2N}}$$

Therefore, to calculate the probability of a collision, one can use:

$$1 - \mathrm{e}^{\frac{(-k)(k-1)}{2N}}$$

Based on the above equation, the probability of at least one collision taking place can be calculated and compared to how the SHA-3 function performs in practice. This is a theoretical value. Hash functions do not work perfectly. Moreover, other factors can play a role in the number of collisions such as the nature of the data and how it maps with that specific hashing function.

For 64-bit hashes, the probability that a collision occurs at datasets as large as $100,000$ is virtually nil. For 32-bit hashes, that probability at $100,000$ names is 0.687, but only 0.011 for $10,000$ names. As expected, collisions are a sure thing at 16-bit hashes for all datasets. This is in line with the results reported in the collision studies previously mentioned.

4.1.4.2 Scalability

To draw conclusions from this data, the growth rate of the number of collisions for a 16-bit hash can be observed as the number of names increases. This is illustrated in the log-log graph shown in Figure 4.2. While 16-bit hashes are too small to provide enough variability for the number of named memory segments being targeted, we can see that the scalability is predictable. In future technologies, the size of resources will continue to increase exponentially, allowing the realization of more and more parallel cores. As the number of cores increases, so does the number of in-flight transactions. The graph shows that the rate of collisions should increase linearly as more transactions are present for future systems, indicating that the results will continue to be relevant as memory subsystems expand.

Moreover, while the AMME grows exponentially, so does the size of the resources on future chips. When viewed this way, the percentage of resource utilization remains constant. This is not true of conventional, reactive caches that have consumed an ever more larger percentage of total resources as the gap between single processor cores and memory latency has increased.

Similar conclusions can be drawn by plotting the data from the SHA-3 core for all the considered hash sizes. Since no collisions were recorded for 64-bit hashes, that data is not included in the graph of 4.3. Collisions begin occurring well before the targeted $10,000$ transactions with 24-bit hashes. The 28-bit hashes offer reasonable security (a collision management system can reasonably cope with the rate of collisions), however, an engineering decision must be taken in regards to the value of the saved resources when opting for 28-bit hashes instead of 32-bit ones. Given that the hardware to correct for duplicate mapping would require a certain amount of resources itself, it may be a wiser choice to select 32-bit hashes to utilize a broader address space.

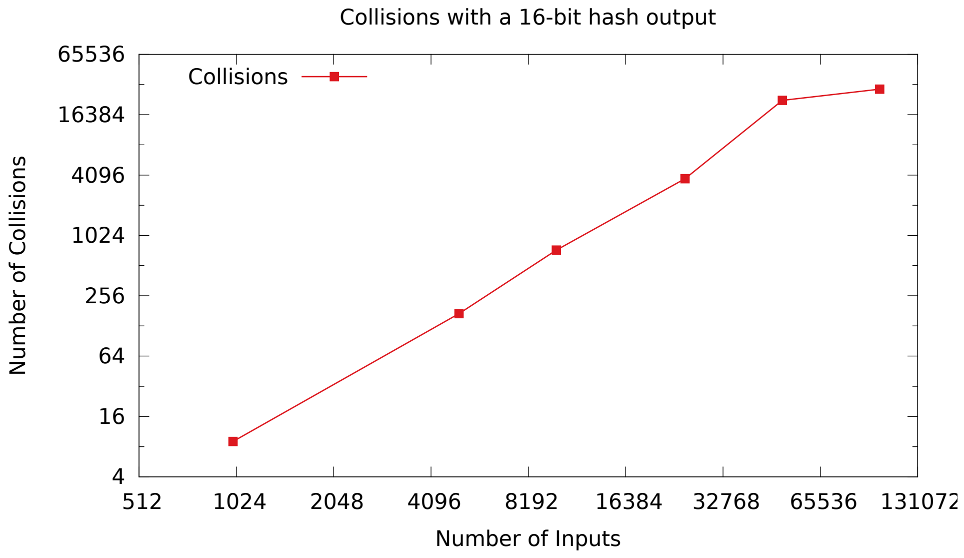


Figure 4.2: Scalability for 16-bit hashes

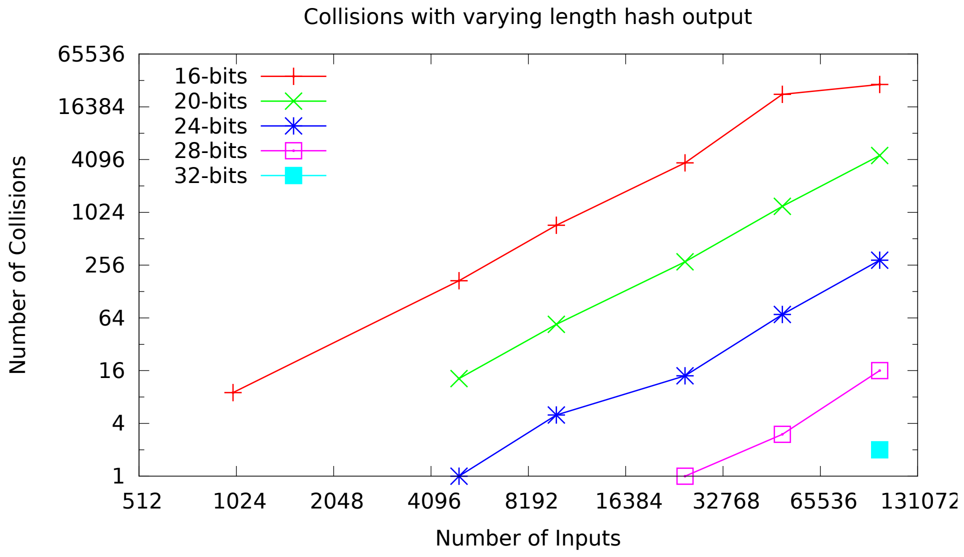


Figure 4.3: Scalability for all hash sizes

Table 4.5: Resource utilization of the Name-to-Key Converter

Number of Slice Registers:	8,239 of 301,440	2%
Number of Slice LUTs:	11,953 of 150,720	7%
Number of Slice LUTS used in Logic:	7,784 out of 150,720	7%
Number of RAMB36E1/FIFO36E1s:	14 out of 416	3%

4.1.5 Name-to-Key Converter Resource Utilization

The design of the B-tree table in the AMME is such that the increase in the key size has a minimal effect on the resource utilization of the chip. The resource utilization of the AMME with different key sizes can be observed in Table 4.5

Since the input to the core was pre-loaded into the on-board DDR3 SDRAM, utilizing a portion of its 128MB large memory space, a significant portion of BRAMs were saved. From the utilization report after placement, it can be concluded that the core itself requires a relatively small percentage of the resources on the FPGA. Additionally, the Virtex-6 is four year old hardware. More recent FPGAs contain double the resources, meaning the footprint of the Name-to-Key Converter would be increasingly less significant. Therefore, it is completely reasonable to use inside the larger memory subsystem design being explored.

The data and analysis from this group of experiments determine *that the implementation of Named DMA using SHA-3 facilitates the functioning of the AMME, and shows that the Named DMA scheme implemented is a scalable solution that will support future large many-core systems.* Additionally, this investigation addresses that scalability problem with named memory segments in software (like LINDA [19]) is overcome by the use of hardware.

4.2 Low-level Parameter Study of the B-tree Hardware Structure

This first group of experiments focuses mainly on the hardware data structure. The main goal of this effort was to implement a B-tree data structure in hardware and run it on an FPGA. As explained in Chapter 1 for large many-core systems, there will be a large number of in-flight messages. Hence, the look-up times of the B-tree

Table 4.6: ML605 development board overview

Memory Technology	Capacity	Number of 4KB blocks	Latency	Bandwidth
BRAM	1849 KB	462	3 cc	13.7 GB/s
DDR3 SODIMM	512 MB	125,000	40 cc	6.4 GB/s

will be critical to the feasibility of the AMME. To do this, we study the number of in-flight messages to estimate the parallelism that AMME will enable.

- Experimental Setup: The experimental infrastructure for this investigation, was a Xilinx ML-605 development board and a Xilinx ZC706 evaluation board. The ML-605 development board has a Virtex-6 XC6VLX240T FPGA and the ZC706 evaluation board has a Zynq-7000 XC7Z045 All Programmable (AP) SoC. Xilinx Embedded Development Kit (EDK) Version 14.5 was used to create and synthesize the design.

- Dataset: For this experiment, the data set is a complete list of files on the Reconfigurable Computing Systems Lab main server. The AMME presented in Chapter 3, handles data that is pre-processed by implementing hashing using a Name-to-Converter core. Thus, the list of filenames were run through the Name-to-Converter core and digests were generated that were used by the AMME.

4.2.1 Resource Utilization for the B-tree Structure

The aim of this investigation is to measure and study of size of the B-tree structure in hardware. Additionally, optimum sizes of the B-tree structure are explored. The resource utilization was observed across two platforms. We started with a size of 4096 and increased the size of the B-tree to observe the chip utilization for larger B-tree structures in the order of 10,000 keys as shown in Table 4.7. Figure 4.4 shows the on-chip resource utilization for B-tree sizes of 1024, 2048, 4096, 8192, 10000, 16384 on the ML605 developer board.

Table 4.7: On-Chip resource utilization of the B-tree structure

Key Size	Number of keys	BRAMs used	Percentage Utilization ML605	Percentage Utilization ZC706
16	4096	13	3%	2.3%
	8192	28	6.7%	5.1%
	10000	57	13.7%	10.4%
	16384	114	27.4%	20.9%
	20000	114	27.4%	20.9%
24	4096	14	3.3%	2.5%
	8192	31	7.5%	5.7%
	10000	61	14.7%	11.2%
	16384	122	29.3%	22.3%
	20000	122	29.3%	22.3%
28	4096	15	3.6%	2.7%
	8192	32	7.7%	5.9%
	10000	63	15.1%	11.5%
	16384	126	30.2%	23.1%
	20000	126	30.2%	23.1%
32	4096	15	3.6%	2.7%
	8192	33	7.9%	6%
	10000	65	15.6%	11.9%
	16384	130	31.2%	23.8%
	20000	130	31.2%	23.8%
64	4096	19	4.5%	3.5%
	8192	42	10%	7.7%
	10000	81	19.4%	14.8%
	16384	162	38.9%	29.7%
	20000	162	38.9%	29.7%

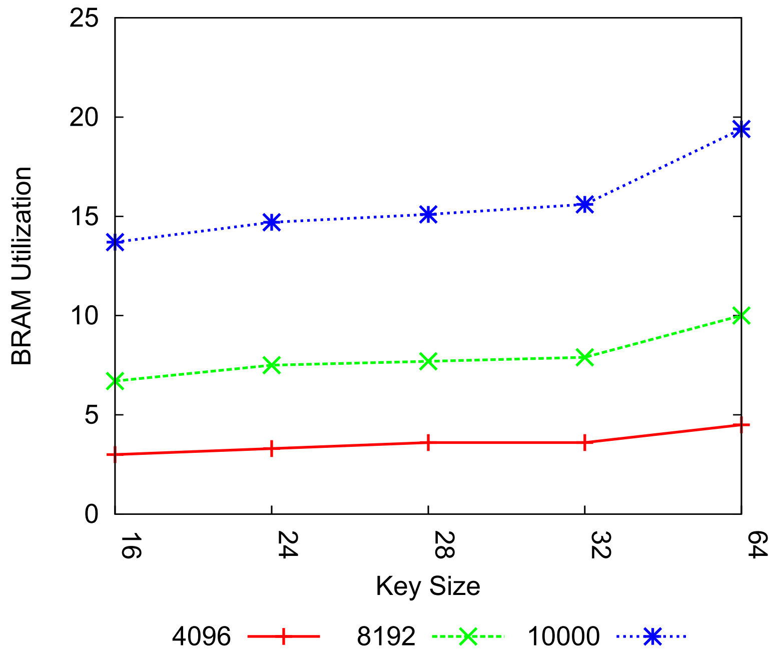


Figure 4.4: BRAM resource utilization for the Xilinx ML605

Figure 4.5 shows the on-chip resource utilization for B-tree sizes of 1024, 2048, 4096, 8192, 10000, 16384 on the ZC706 Evaluation board.

As observed from figures 4.4 and 4.5, the hardware B-tree structure was configured for different sizes and orders to observe resource utilization. Two main results are observed in this experiment. The first one is that the trend of resource utilization of the AMME is maintained on a different, bigger FPGA. This is important as it shows the capability of the AMME to maintian low resource utilization on a newer platform. This enables us to increase the size of the B-tree and support more keys in the system. Another key result observed here, is that on an ML605 board, the AMME utilizes less than 20% of the total resources on chip. On the ZC706 evaluation board, the AMME utilizes *less than* 15% of the total resources on chip. This is a huge advantage over existing memory subsystem that consume anywhere between 60% to 90% of the total on-chip resources. This shows that implementing the AMME can free up close to 80% of the on-chip resources which can then be used for computation. This shows that the AMME will support a higher degree of parallelism in a large many-core system.

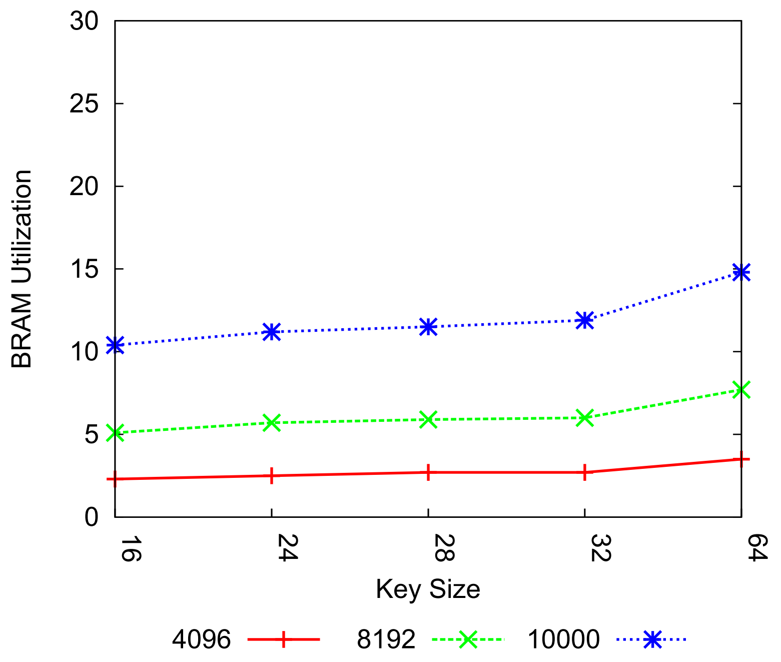


Figure 4.5: BRAM resource utilization for the Xilinx ZC706

4.2.2 Depth and Order of the B-tree

Size is an important constraint to consider while designing a structure that holds the metadata of a system. As with any tree data structure, we analyze B-tree characteristics (the depth and the order of the tree) to observe how the resource utilization and the access times were affected. The orders chosen were 4 (each B-tree node has four keys), 8 (each node has seven keys), and, 12 (each node has eleven keys). The order of the tree changes the depth for a given number of nodes. Figure 4.6 shows the change in the depth of the tree with regard to the other of the tree.

4.2.3 Latencies of Operations in Isolation

While the operations of a memory controller are always READ and WRITE, the AMME consists of internal operations that are more in tune with the working of a B-tree. Broadly, they are *add, find, update, traverse, mark_for_delete, split.* These operations are explained in detail in Chapter 3.

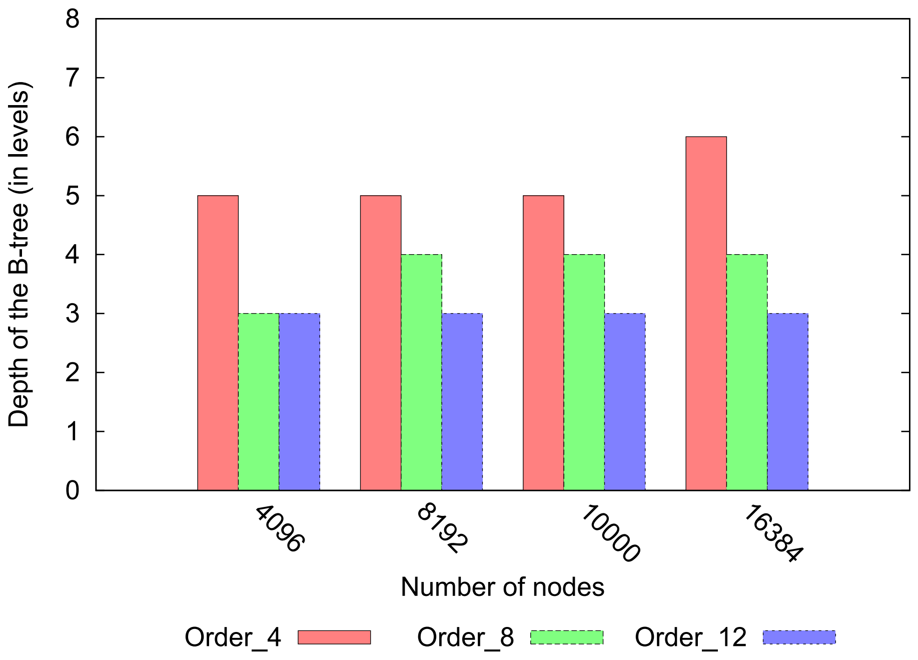


Figure 4.6: B-tree characteristics: Order and Depth

Table 4.8: Clock cycles for each operation: B-tree version 2

Operation	Cycles per level
ADD	6
FIND	2
TRAVERSE	$8 \times d$
VALIDATE	$4 + d$
SPLIT	$30 \times d$
MARK FOR DELETE	$4 + d$

Table 4.9: Clock cycles for each operation: AMME with Order 4 B-tree

Operation	Cycles per level
ADD	2
FIND	2
TRAVERSE	$6 \times d$
VALIDATE	$1 + d$
SPLIT	$23 \times d$
MARK FOR DELETE	$1 + d$

Table 4.10: Clock cycles for each operation: AMME with Order 8 B-tree

Operation	Cycles per level
ADD	3
FIND	2
TRAVERSE	$4 \times d$
VALIDATE	$1 + d$
SPLIT	$22 \times d$
MARK FOR DELETE	$1 + d$

4.2.3.1 B-tree Cache System Latencies

Table 4.8 shows the time (in clock cycles) for the basic operations. The depth of the tree, d, is an important factor to these results. The *Traverse* operation is directly proportional to the depth of the tree, specifically, $8 \times d$. The *Validate* operation takes 4 clock cycles and additionally adds one clock cycle per unit increase in depth. This does not apply to the *Add* and *Find* operations as the traversal points the controller to the right place, after which only the new key needs to be added, or in the case of *Find*, returned. The split operation is the most expensive, especially if it results in multiple splits. The time measured for a split is 30 clock cycles.

4.2.3.2 AMME Latencies

In the newer version of the system, the BRAMs were widened enough to hold an entire node of the B-tree as opposed to just one key. This greatly helped reduce the look up times for the B-tree. This can be observed in Tables 4.9, 4.10 and 4.11.

Figure 4.7 shows the latencies of the operations in isolation. The latencies for

Table 4.11: Clock cycles for each operation: AMME with Order 12 B-tree

Operation	Cycles per level
ADD	3
FIND	2
TRAVERSE	$4 \times d$
VALIDATE	$1 + d$
SPLIT	$19 \times d$
MARK FOR DELETE	$1 + d$

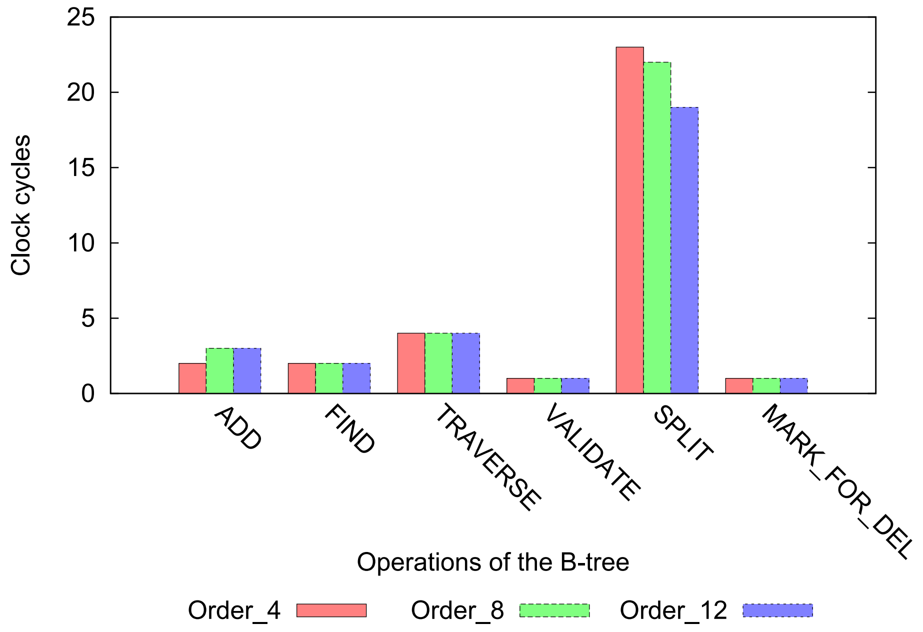


Figure 4.7: Latencies of the operations of the B-tree

Table 4.12: Access times for order 4, order 8, and order 12 B-tree

Case	Number of nodes	Order 4	Order 8	Order 12
Best Case	4096	50 ns	43.20 ns	43.20 ns
	8192	50 ns	43.20 ns	43.20 ns
	10000	50 ns	43.20 ns	43.20 ns
	16384	50 ns	43.20 ns	43.20 ns
Worst Case	4096	190 ns	92.34 ns	92.34 ns
	8192	190 ns	129.42 ns	92.34 ns
	10000	190 ns	129.42 ns	92.34 ns
	16384	225.4 ns	129.42 ns	92.34 ns

the split operation are the highest. But the way a B-tree works, we can accurately predict the largest number of splits that can occur, given a particular set of nodes. This maintains a degree of predictability that is very useful when scaling this design.

4.2.4 Access Times

Theoretically, the search time for a B-tree is $O(logn)$, where n is the total number of nodes in the B-tree. In this investigation, this is compared with the actual measured access times for the B-tree hardware data structure. This was then used to emulate accesses to the memory. The average access time was observed for a tree of depth n, and recorded. The best case and worst case access times for different orders and sizes of the B-tree are shown in Table 4.13.

Thus the absolute best case access time is 43.2 ns for a key that residing in the first level on the order 8 or 13 B-tree, and the absolute worst case access time was 225.4 ns, for a B-tree with 16,384 nodes. The worst case time was when the B-tree was fully populated and the key is fetched from the *leaf* layer.

Another measurement in this experiment was the worst case time to add a node to the B-tree. This is measured when the tree is populated in such a way that adding a node will result in multiple splits all the way to the root of the tree.

The main observation from these experiments is that the hardware B-tree data structure maintains the trends observed in the theoretical calculations. This brings

Table 4.13: Access times for order 4, order 8, and order 12 B-tree with splits

Case	Number of nodes	Order 4	Order 8	Order 12
Best Case	4096	170 ns	170 ns	150 ns
	8192	170 ns	170 ns	150 ns
	10000	170 ns	170 ns	150 ns
	16384	170 ns	170 ns	150 ns
Worst Case	4096	770 ns	440 ns	390 ns
	8192	770 ns	575 ns	390 ns
	10000	770 ns	575 ns	390 ns
	16384	920 ns	575 ns	390 ns

a measure of *predictability* to the hardware data structure that is much needed for memory subsystems in larger many-core systems.

The data collected from this group of experiments determine *that the implementation of the hardware data structure is feasible and that the measured access times of the data structure are comparable to theoretical observations.* This is a crucial result of this research. Additionally, this set of experiments also show the use of a named DMA model for the memory subsystem. The next subsection presents a detailed analysis on the different aspects of the AMME as a whole.

4.3 AMME Results

The next goal of this research is to test the B-tree architecture with different memories like Block RAM and DDR3 SDRAM. Similar to most of the systems being developed today, the objective of the AMME is also to try to implement an enormous amount of on-chip bandwidth. But the real test is to impose a hierarchy on the system thereby introducing architecture bottlenecks.

The system presented and explained in Chapter 3 has been implemented on a Xilinx ML-605 development board with a Virtex-6 (XC6VLX240T) FPGA. Version 14.5 of the ISE/EDK tools were used to create and synthesize the design. The test infrastructure consists of a MicroBlaze core on the ML605 FPGA board will run processes that randomly request data from the AMME. The load generated on the

AMME is synthetic, but the data itself (hashed digests) is derived from the Amazon reviews dataset. The Amazon reviews dataset are be hashed to create 28-bit keys for the B-tree and the content of the reviews will the data to be stored in the on-chip and off-chip memories. The capacity of memory required to prevent thrashing is observed. The AMME has multiple channels to the memory (link to on-chip BRAM, link to off-chip DDR3). The main aim of this investigation was to observe that all the memory channels were exercised efficiently and to measure the time to completion of higher level commands in the system. This phase of validation also provides a better understanding on how to proportion many-core chips in a way that will improve memory performance without being detrimental to processor performance.

4.3.1 Time to Completion

The access times measured in Section 4.2.4 are exclusively for the operations of the B-tree. Another metric that was measured was the average time to perform a READ operation and a WRITE operation. This was done at a higher level than the previous experiment. The AMME was configured to have an order 8 B-tree with a total size of 10,000 keys. The hashed key size is 28, and is derived from the Amazon reviews dataset. A stream of WRITE and READ requests were injected into the system and the total time to perform these operations were observed. In this investigation, we measure the time to completion for the AMME for a READ command, WRITE command and the time to completion for a Delete (performing an epoch). Also, for the purpose of this experiment, the payload is 4KB blocks of data, in order to emulate the Green/White Architecture requirements. A modified synthetic load generator was used to initiate data transfer.

4.3.1.1 Average Time for READ and WRITE Operations

Since in the context of this dissertation, the AMME has two channels to different memories that vary in latency (as dictated by the available hardware), the best case time and the worst case time to perform these operations were measured. The

Table 4.14: Time to perform READ/WRITE considering 4KB blocks of data

Case	Operation	Size of the Order 8 B-tree	Time
Best Case	READ	10000	2.0352 μS
	WRITE	10000	2.162 μS
Worst Case	READ	10000	12.3614 μS
	WRITE	10000	12.807 μS

Table 4.15: Time to perform an Epoch or Delete

Size of the B-tree	Time
4096	370.94 μS
8192	740.48 μS
10000	903.91 μS

measured are times shown in Table 4.14.

It is observed that these times are indeed very small and thus reasonable. When compared to a system with a regular memory controller, this result is favorable.

4.3.1.2 Average Time for an Epoch

The epoch controller of the AMME was tested to perform a DELETE operation. Table 4.15 shows the time taken to perform delete on different B-tree sizes.

The time to perform an epoch for a B-tree with 10,000 keys was observed to be around 904 μS which is very small. Unfortunately, there is no direct point of reference at this point, although it is clear that this number is favorable.

4.3.2 Energy Expended

The amount of energy spent in moving data is directly proportional to the distance moved. This is an increasing concern in large many-core systems. This experiment is an analytical study from existing numbers. Measuring the energy expended is out of the scope of this dissertation. Many authoritative sources record the specific cost of data movement from different memories in terms of energy [45], [22]. The energy cost described by Jacob et al can be seen in Table 4.16.

Using the B-tree mechanism explained in Chapter 3, the AMME ensures two

Table 4.16: Energy per access for memory technology

Technology	Energy per Access
On-chip cache	1 nJ
Off-chip cache	10–100 nJ
DRAM	1–100 nJ (per device)
Disk	100–100 mJ

things: *only* the required data is moved; and unless absolutely necessary the data is never moved more than once. The channels to the green core scratchpads from the AMME enable direct DMA to and from the scratchpads, thus moving the data only when required. This shows that the AMME can support *explicit* data movement and thus reduce the energy expended by moving only necessary data.

CHAPTER 5: CONCLUSION

Steady increases in the density of Integrated Circuits (IC) has resulted in more transistors per chip and faster transistors. Unfortunately, memory technology has not seen similar improvements. Moreover, technology trends are likely to exacerbate this problem. With Exascale architectures on the horizon, a lot of complications that necessitate branching out from the norm of conventional computer architecture research are exposed. With the ever-increasing processor clock rates, the performance of a system is not just determined by the amount of parallelism available and achieved by a system, but mostly by the ability of the system to keep its parallel hardware operational. An expected bottleneck to this is memory. The growth of memory technology has been stagnant over the past few decades which had led to little or no improvement in memory latency times. New methodologies are required to improve memory latency and increase memory bandwidth.

In this dissertation, a novel memory subsystem, the Active Memory Management Engine, is presented. It uses a B-tree based mechanism with a flatter hierarchy to effectively manage multiple channels to different types of memory. It accomplishes this by proactively managing a large number of outstanding requests in such a way that the channel to external memory is utilized effectively. To be successful, it imposes a task-oriented computational model on computational cores and requires them to multi-task in order to generate multiple, overlapping memory requests that can be served by the memory subsystem in any order. Using Little's Law, we propose that such an organization will create a stochastic system at the architecture level that hides memory latency, increases the effective bandwidth of an off-chip memory channel, and increases the task completion rate for memory-bound systems. Additionally, a priority scheme is implemented that gives the the memory subsystem the ability to re-order the tasks to always keep the cores busy with computation. The end goal is to enable the memory subsystem to handle a wide range of memory technologies that vary in capacity, bandwidth, and latency.

This approach is not appropriate for all settings. In particular, many FPGA-based designs have an enormous degree of parallelism but are not memory bound. It may be the case that there are designs that interface to instruments, sensors and actuators, that provide a continuous source or sink of information. In other cases there are applications that are very computationally intensive and have a relatively small state. Nonetheless, there is a large range of applications in the High-Performance Computing domain (from graph algorithms to iterative algorithms) that fit the proposed model.

The design process of the AMME architecture started with the theoretical formulation of a stochastic model to validate the idea of implementing this new and unconventional architecture. Then, a proof of concept design was implemented on an FPGA for testing the feasibility of implementing a data structure in hardware. Several new concepts are presented in the final design of this new architecture. A wide range of experiments were performed to evaluate this design. The B-tree mechanism of the AMME enabled a predictable access time for memory segments. Also, the resource utilization of the AMME was under 20% of the total resources of the hardware platform. This allows more resources to be allocated for computation and thus enables a *higher degree of parallelism.* A new addressing scheme was also introduced in this dissertation that uses named memory segments instead of traditional byte addressable memory. This capability of the AMME to perform *named DMA* improves the scalability of the memory subsystem. The access times measured are an order-of-magnitude faster than conventional deep hierarchy caches or memory systems. The AMME *actively* manages the data thus enabling explicit data movement.

The aim of this research is analyze the memory subsystem in a many-core environment. The Green/White architecture is presented in this dissertation to provide a context for the AMME subsystem. The overall Green/White architecture includes a PyDac framework [16], a radical interconnection network [20] and the *Active Memory Management Engine*. This dissertation proves, theoretically and empirically, that the active memory subsystem *will not be a bottleneck in a many-core architecture.*

REFERENCES

[1] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," 1965.

[2] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on.* IEEE, 2011, pp. 365–376.

[4] F. J. Pollack, "New microarchitecture challenges in the coming generations of cmos process technologies (keynote address)," in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture.* IEEE Computer Society, 1999, p. 2.

[5] P. Kogge, et al., "Exascale computing study: Technology challenges in achieving exascale systems," DARPA Information Processing Techniques Office (IPTO) sponsored study, Tech. Rep. TR-2008-13, 2008. [Online]. Available: www.cse.nd.edu/Reports/2008TR-2008-13.pdf

[6] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[7] C. Kozyrakis and D. Patterson, "A new direction for computer architecture research," *Computer*, vol. 31, no. 11, pp. 24 –32, nov 1998.

[8] D. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, 1999, pp. 248 –259.

[9] J. R. Johnson, "Automated performance tuning," in *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, ser. PASCO '10. New York, NY, USA: ACM, 2010, pp. 20–21. [Online]. Available: http://doi.acm.org/10.1145/1837210.1837215

[10] G. Liao, X. Zhu, and L. Bhuyan, "A New Server I/O Architecture for High Speed Networks," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, Jan. 2011, pp. 1–11.

[11] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference.* ACM, 2007, pp. 746–749.

[12] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint*

computer conference, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[13] S. Holzer, "Optimization for Enhanced Thermal Technology CAD Purposes," Ph.D. dissertation, Technischen Universitt Wien, Vienna, January 1976.

[14] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/MC.2008.209

[15] D. Bertsimas and D. Nakazato, "The distributional little's law and its applications," *Operations Research*, vol. 43, no. 2, pp. 298–310, 1995.

[16] B. Huang, "A distributed runtime system that bridges heterogeneous many-core architecture with divide-and-conquer programming paradigm (in progress)," Ph.D. dissertation, University of North Carolina, Charlotte, 2014.

[17] R. Bayer and E. McCreight, *Organization and maintenance of large ordered indexes.* Springer, 2002.

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[19] N. Carriero and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[20] W. V. Kritikos, "The impact of routing on the resilience of large network on chip systems (in progress)," Ph.D. dissertation, University of North Carolina, Charlotte, To be published.

[21] R. Sass and A. G. Schmidt, *Embedded systems design with platform FPGAs: principles and practices.* Morgan Kaufmann, 2010.

[22] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk.* Morgan Kaufmann, 2010.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to algorithms.* MIT press Cambridge, 2001, vol. 2.

[24] P. Marchal, "Field-programmable gate arrays," *Commun. ACM*, vol. 42, no. 4, pp. 57–59, 1999.

[25] Xilinx, "Products and services, intellectual property," March 2008.

[26] Xilinx, Inc., "ML605 hardware user guide," October 2012.

[27] Y. Rajasekhar and R. Sass, "A first analysis of a dynamic memory allocation controller (dmac) core," in *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, ser. SAAHPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 64–67. [Online]. Available: http://dx.doi.org/10.1109/SAAHPC.2011.23

[28] D. Knuth, "The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching," 1968.

[29] J. Rice, W. Osborn, and J. Schultz, "Implementation of a spatial data structure on a fpga," in *Advances and Innovations in Systems, Computing Sciences and Software Engineering.* Springer, 2007, pp. 207–210.

[30] V. Sklyarov, I. Skliarova, R. Oliveira, D. Mihhailov, and A. Sudnitson, "Processing tree-like data structures in different computing platforms," in *Proc. Int. Conf. on Informatics and Computer Applications-ICICA2011*, 2011, pp. 112–116.

[31] M. Chrzanowska-Jeske, Z. Wang, and Y. Xu, "A regular representation for mapping to fine-grain, locally-connected fpgas," in *Circuits and Systems, 1997. ISCAS'97., Proceedings of 1997 IEEE International Symposium on*, vol. 4. IEEE, 1997, pp. 2749–2752.

[32] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[33] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge, "An analytical model for designing memory hierarchies," *Computers, IEEE Transactions on*, vol. 45, no. 10, pp. 1180–1194, 1996.

[34] C. Chow, "Determination of cache's capacity and its matching storage hierarchy," *Computers, IEEE Transactions on*, vol. 100, no. 2, pp. 157–164, 1976.

[35] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 2, pp. 184–215, 1989.

[36] A. J. Smith, "Line (block) size choice for cpu cache memories," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 1063–1075, 1987.

[37] P. J. Denning, "The working set model for program behavior," *Communications of the ACM*, vol. 11, no. 5, pp. 323–333, 1968.

[38] J. Dennis, G. Gao, and X. Meng, "Experiments with the fresh breeze tree-based memory model," *Computer Science - Research and Development*, vol. 26, no. 3-4, pp. 325–337, 2011. [Online]. Available: http://dx.doi.org/10.1007/s00450-011-0165-1

[39] G. Liao, X. Zhu, and L. Bnuyan, "A new server i/o architecture for high speed networks," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on.* IEEE, 2011, pp. 255–265.

[40] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge, "End-to-end performance forecasting: finding bottlenecks before they happen," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 361–370, 2009.

[41] D. Burger, J. R. Goodman, and A. Kägi, *Memory bandwidth limitations of future microprocessors.* ACM, 1996, vol. 24, no. 2.

[42] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, "Leap scratchpads: automatic memory and cache management for reconfigurable logic," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays.* ACM, 2011, pp. 25–28.

[43] E. S. Chung, J. C. Hoe, and K. Mai, "Coram: An in-fabric memory abstraction for fpga-based computing," in *Nineteenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2011.

[44] S. Kim and C. H. Lam, "Transition of memory technologies," in *VLSI Technology, Systems, and Applications (VLSI-TSA), 2012 International Symposium on.* IEEE, 2012, pp. 1–3.

[45] S. Swanson and A. M. Caulfield, "Refactor, reduce, recycle: Restructuring the i/o stack for the future of storage," *Computer*, vol. 46, no. 8, pp. 52–59, 2013.

[46] D. Gross and C. M. Harris, *Fundamentals of queuing theory*, 1998.

[47] B. Huang and R. Sass, "Leveraging python in a high-level programming model based on divide-and-conquer strategy," in *Sixth International Workshop on Parallel Programming Models and Systems Software for High-end Computing - Under review*, 2013.

[48] S. Hawayek, "Feasibility study of using named memory segments instead of byte addressable memory in highly parallel many core systems," Ph.D. dissertation, University of North Carolina, Charlotte, 2014.

[49] M. P. Guido Bertoni, Joan Daemen and G. V. Assche, "The Keccak sponge function family," url: http://keccak.noekeon.org/files.html.

[50] Y. Rajasekhar and R. Sass, "A novel memory subsystem and computational model for parallel reconfigurable architectures," in *Euro-Par 2013: Parallel Processing Workshops.* Springer, 2014, pp. 444–453.

[51] J. Leskovec, "Stanford large network dataset collection," *URL http://snap. stanford. edu/data/index. html*, 2011.

[52] M. Peyravian, A. Roginsky, and A. Kshemkalyani, "On probabilities of hash value matches," *Computers & Security*, vol. 17, no. 2, pp. 171–176, 1998.