SOFTWARE MEMORY CONTROLLER DESIGN : ISSUES AND CHALLENGES

by

Aditi Gaur

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2015

Approved by:

Dr. Ron Sass

Dr. James M. Conrad

Dr. Bharat Joshi

ABSTRACT

ADITI GAUR. Software memory controller design : issues and challenges.
(Under the direction of DR. RON SASS)

As the memory wall becomes an obstacle in the multi-core architecture, many hardware solutions have been proposed to utilize the available memory bandwidth more efficiently. One such implementation uses a B-tree based controller implemented in hardware to manage a memory subsystem. This thesis investigates various design factors and design decisions to implement the B-tree controller purely in software running on a generic soft-core or a hard-core processor. It investigates why software is slower and what factors if changed, can make it a feasible solution, it at all, and finally present an environment in which software can become competitive to hardware.

ACKNOWLEDGMENTS

A lot of people helped along the way from start to the finish of this Masters Thesis contributing in many different ways. I would sincerely like to thank my advisor, Dr. Ron Sass for his guidance throughout my Masters and for this work. It was a great experience working under him. I would like to thank my committee members Dr. Joshi and Dr. Conrad for all their support. I would also to thank my Mom, Dad. Nothing would be possible without you all being there for me like my rock. And of course to Abby, Priyam for being great friends and always talking me out of stressful situations. To all my colleagues in RCS lab. It was great working with you all. And at last but far from least, to Varun for running around so much for me and being there when needed the most. This acknowledgment justfies only a small fraction of gratitude I feel for you all.

TABLE OF CONTENTS

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER 1: INTRODUCTION 1

1.1 Memory Wall 1

1.2 Memory Controller Design 2

1.3 The Design Space 2

1.4 Thesis Question 4

1.5 Outline 5

CHAPTER 2: BACKGROUND 6

2.1 Memory Hierarchy 6

2.2 Active Memory Subsystem 7

2.3 Concept of a B-tree 9

CHAPTER 3: DESIGN 14

3.1 Hardware 14

3.1.1 MicroBlaze 14

3.1.2 Zynq Processing System 15

3.1.3 On-Chip Memory 16

3.1.4 External (Off-chip) Memory 18

3.1.5 Peripherals 19

3.2 Software Setup 20

3.2.1 Implementation 21

3.2.2 Compiler and Linker 23

3.2.3 Datasets 24

CHAPTER 4: EVALUATION 25

4.1 Initial Evaluation 25

4.1.1 Dataset Variation 31
4.2 Improving Latency 32
4.3 Improving Speed 35
4.4 Final Evaluation 39
CHAPTER 5: CONCLUSION 44
REFERENCES 46

LIST OF TABLES

TABLE 2.1: Time complexity of various operations in a B-tree 11

TABLE 2.2: Best case heights of a B-tree with the given number of keys 13

TABLE 2.3: Worst case heights of a B-tree with given number of keys 13

TABLE 4.1: Lookup access times using the Amazon data set 26

TABLE 4.2: Lookup access times using server data set 27

TABLE 4.3: Time to insert a key from Amazon dataset 28

TABLE 4.4: Percentage of execution time 30

TABLE 4.5: Resource utilization 30

TABLE 4.6: Variation in average lookup time (microseconds) with dataset size 32

TABLE 4.7: Variation in average time to insert 500 and 5000 keys 32

TABLE 4.8: Lookup access times using(500 keys) on-chip memory 33

TABLE 4.9: Resource utilization with most accesses restrained to on-chip 34

TABLE 4.10: Time(microseconds) to insert keys in on-chip B-tree 35

TABLE 4.11: Lookup times(nanoseconds) on ARM with off-chip B-tree 36

TABLE 4.12: Time (ns) to insert in off-chip B-tree on Zynq PS 36

TABLE 4.13: Lookup times (ns) on ARM running at 800Mhz with off-chip B-tree 38

TABLE 4.14: Lookup times (ns) on ARM at 800Mhz and on-chip B-tree 39

TABLE 4.15: Time to insert in on-chip B-tree on ARM core at 800 Mhz 39

TABLE 4.16: Resource utilization on Zynq-PS 41

LIST OF FIGURES

FIGURE 2.1:	High level block diagram of a Green-Core	8
FIGURE 2.2:	An example of insertions in a 2-3-4 B-tree with preemptive splits	12
FIGURE 3.1:	Hardware data path	18
FIGURE 4.1:	Lookup access times of the B-tree using Amazon data	27
FIGURE 4.2:	Lookup access times using server data set	28
FIGURE 4.3:	Lookup access times of on-chip B-tree	34
FIGURE 4.4:	Lookup times on ARM with off-chip B-tree	37
FIGURE 4.5:	Improvement due to hiding latency and increasing speed	38
FIGURE 4.6:	Comparing lookup times in Table 4.11 and Table 4.14	40
FIGURE 4.7:	Comparing insert times in Table 4.12 and Table 4.15	40
FIGURE 4.8:	Comparison of B-tree access times on hardware vs software	41

CHAPTER 1: INTRODUCTION

Understanding the design decisions involves taking into account various factor that contribute to an optimal design. The decision to implement a certain functionality in hardware or software is commonly referred to as Hardware-Software Co-Design. Often common assumptions can be counter intuitive. A motivating example in this regard is the concept of memory wall.

1.1 Memory Wall

Memory hierarchy as applied to a multi-core environment raises new challenges as it tries to solve the old ones. The tried and tested approach towards hiding memory latency imposes severe cost paid by wasted memory bandwidth, extensive data movement and consequent energy considerations. These issues lead us to an important roadblock in multi-core architecture namely the Memory Wall [1].

Highly parallel architectures have high bandwidth requirements and the illusion of hiding latency is often not enough. Usage of caches, offer disadvantages of excessive data movement, and increased demands on energy. Extra circuitry used to handle cache hits and misses as well as replacement becomes too complex in a multi-core environment. Bandwidth is inefficiently utilized as atomic unit of transfer in a cache is only the cache line, which typically is much lesser than memory bandwidth available. The passivity of caches lies in the fact that it cannot effectively use the available bandwidth [2], especially when it doesn't need to. For example in case of a cache miss, it might make an off chip access. But with all subsequent hits, it will not do any useful work to maximize the use of available bandwidth. For multi-core architectures this could mean that off chip accesses could be made in bursts of cycles where all cores are starving for bandwidth, whereas in other cycles bandwidth remains

completely unused. Thus the use of memory bandwidth is non uniform. Simple data caches thus employ a short term strategy, with no long term goal in sight. For such architectures, the performance hence becomes memory bound [2]. In this environment, many researchers have consensus for the need of a more flatter hierarchy that optimizes data movement and use bandwidth more efficiently, thus employing a more long term strategy and not making the cores starve for data at any point.

1.2 Memory Controller Design

A novel hardware memory controller design was proposed as part of another study [3] that satisfies the need of a flatter hierarchy, while achieving low latency in operations in the backdrop of a multi-core architecture. The controller is fully implemented in hardware, with a B-tree based controller for managing the pool of data that can be looked up using keys. The hardware implementation of the B-tree uses on-chip resources (BRAM's, LUT's) to manage large payloads to various cores. The main aim is to not only provide low latency flatter hierarchy, but to also manage the external memory channels more effectively. This implementation serves to prove that the hardware design of the controller is feasible and will not limit the performance of the multi-core architecture. The hardware B-tree model access times are very close to theoretical access times, which further increases the scope of this design.

1.3 The Design Space

Hardware/Software co-design involves making choices between implementing specific functionality in hardware (i.e HDL, or using dedicated hardware support) or software (C++ or similar) [4]. A common assumption is that hardware is always faster than software and that the fundamental engineering trade-off is speed versus resource utilization. While true, in actuality design space is much more complex. Often with reconfigurable devices such as an FPGA, the maximum clock frequencies possible for the processor and custom hardware are drastically different, sometimes by one or two orders of magnitude different. There are many other subtle differences

beyond just clocking speeds. For example some algorithm (or data structure) may produce different memory access patterns, have different overhead delays, access or ignore on-chip resources, may or may not use the hardware efficiently. This renders navigating the design space as fairly complex and specific to the needs of the application.

To better understand the nature of what design decisions have the most impact, we studied the B-tree data structure with the core operations of insert and lookup. Specifically since the core of the hardware memory controller implemented on an FPGA described above relies on the B-tree controller for its main set of operations. The question we asked was, how would a software-only implementation (using an on-chip soft or hard processor) fare in memory controller design? Can we simply transfer some or most of the hardware functionality to the software and instead keep the hardware generic? With a goal of managing 10,000 to 40,000 key-value pairs, this requires a detailed analysis of the trade-off involved and specifically where exactly we pay the price.

We anticipated that the software would be slower but by quantifying how slow exactly and characterizing the root causes would be valuable for future co-designers for applications that use B-tree or related data structures. We want to know if improvements are even worth trying for and if yes, then what are the upper limits. How far can we really push the envelope to deterministically answer this question. What we learned was that the performance of a conventional (soft)processor-bus-memory design was far worse than expected and almost no application would benefit from a software only solution given that a hardware B-tree based solution is readily available. While we were using much less on-chip resources and the resources that are used are highly generic and not custom to the application which also leaves a window for making the software hardware independent. But if the software is extremely slow, then low resource utilization cannot be the selling point.

However upon further investigation in identifying what exactly makes the software slower we could pin-point on many varied performance restricting factors. Analyzing and optimizing each factor in isolation as well as in combination makes the design decisions more complex than imagined. While optimizing some factors resulted in only limited improvements, but working on other factors offered a number of improvements that had a significant ability to close the gap between hardware and software implementations. In certain situations, software was found to be very competitive with the hardware.

1.4 Thesis Question

The question we want to ask is this- We know that software is expected to be slower than hardware. We want to know *why?*. We want to know the root cause and if it can be eliminated by modern software and hardware solutions. Understanding the root cause can lead us in various directions such as- is it cache misses? Is a cache miss too expensive and can that penalty be made better? Is it the clock speed, or the pipeline throughput? Is the sluggishness dominated by off-chip accesses? Do these factors contribute equally and does improving one factor worsen the other or improve the other? Namely do factors affect each other?

Which brings us to the main question- *Is there any one factor that, if changed, would yield significantly better results, favoring a software solution instead? If not, are there any combination of factors if improved can yield a favorable result? This thesis explores various reasons why software might fail to yield desired results and whether modifying any factor would significantly impact the result or not*

To answer this question we performed a set experiments that follow an elimination policy. Our end goal here is to get results that are comparable to hardware and analyze carefully the impact of each design change we go on to make. We first perform a very basic initial experiment to get the upper bound of how software performs at the very basic level with no optimizations. We then attempt to isolate certain factors, apply

a change in those factors in isolation and study the improvement if any. Finally after carefully studying the impact of each factor in isolation, we combine all the studied factors into a final evaluation and the impact of the combination of varied factors is then analyzed.

1.5 Outline

The remaining of this thesis is organized as follows. The chapter 2 provides some necessary background required to understand the role of a memory controller in a multi-core architecture as well as how such a memory controller design can be implemented using B-trees. It introduces B-trees conceptually, and how they are created, how keys are inserted and how lookups are performed. How height of the tree impacts the worst case situations is also explained thoroughly. Chapter 3 explains the basic hardware design that was set up on the FGPA to conduct the experiments. It introduces various design components such as soft-core and hard-core processors and latencies in various situations. Also it explains the software implementations and the algorithms used. Chapter 4 presents step by step each of the experiments performed, along with its experimental setup and the results obtained from each experiment and what those results mean in terms of design decisions. Chapter 5 finally provides the concluding arguments for this thesis.

CHAPTER 2: BACKGROUND

2.1 Memory Hierarchy

Memory hierarchy became important as computing power of processors continued increasing and consequently the demands of a faster and cheaper memory. For a single compute core model employing sequential operations, this performance gap between memory and processing could easily be reduced by employing layers of faster memory to hide the latency of the slowest memory. This proved to be a viable solution that could hide the expensive cost of clock cycles needed to fetch data from off-chip memory. The main idea is to keep the cost per byte of the slowest memory and speed almost as fast as the fastest memory [5]. Thus processor sees a low latency cache followed by complex circuitry to control the data movement between the lower levels and maintain consistency in case of writes. Each cache line size is either replaced or fetched from the internal main memory or the lower level of cache. A similar system is employed by the main memory to fetch pages of data from the external memory. This involves a lot of data movement and considerable use of memory bandwidth and energy consumption [6]. Not only that, it consumes majority of on-chip resources reaching in worst cases, a utilization of up to 90% [7]. These issues were still ignorable compared to the advantages it provided by hiding the memory latency in single-core computing with focus on compute-bound performance advantage. Recent advancements in computing architecture have shifted focus towards multi-core highly parallelized computing, due to increasing performance requirements of modern systems. However modern multi-core computing systems are not subject to the same advantages. Factors that were ignorable earlier, are significant now [8].

2.2 Active Memory Subsystem

The ideas presented in this thesis are in the background of a novel multi-core computation model, i.e a Green-White core architecture, that takes into account the problems mentioned above and strives for best optimal use of memory bandwidth and avoiding unnecessary data movement to save power. It makes use of heterogenous multi-core environment where all cores are not expected to perform equally. Green cores are simple processors that have access to bytes of addressable multiple scratchpad memories which departs from the conventional style of hierarchy. These processors are not complex and are relatively simple with reduced pipeline depth, lower clock rate, and reduced silicon. Their main selling point is that they are power efficient and light on resource utilization. They are especially tailored towards applications and are designed with the programmers perspective in mind. We also have complex compute cores, which are the white cores that are capable of complex calculations with full support for resilience. The white cores follow the conventional style of memory hierarchy and are the workhorse for critical sequential applications. This provides us a heterogenous computing model, where none of the processors are required to over-perform while none underperforming either. Each processor handles, only what it can do best.

Green cores are connected via an on-chip network to an active memory subsystem that is responsible for supplying the data to them. Active memory management engine, AMME [3] manages all the off-chip accesses (by having multiple channels to memory) and makes sure that any of the cores are never starved for data. It has capability to supply data requested by a core, while the data is being computed by another. It hides latency while efficiently managing bandwidth. The processor views the multiple chunks of scratchpad memory as essentially flat, which is very similar to how a processor would view a conventional cache. AMME also manages all data transfers, between scratchpads and memory subsystem directly via DMA

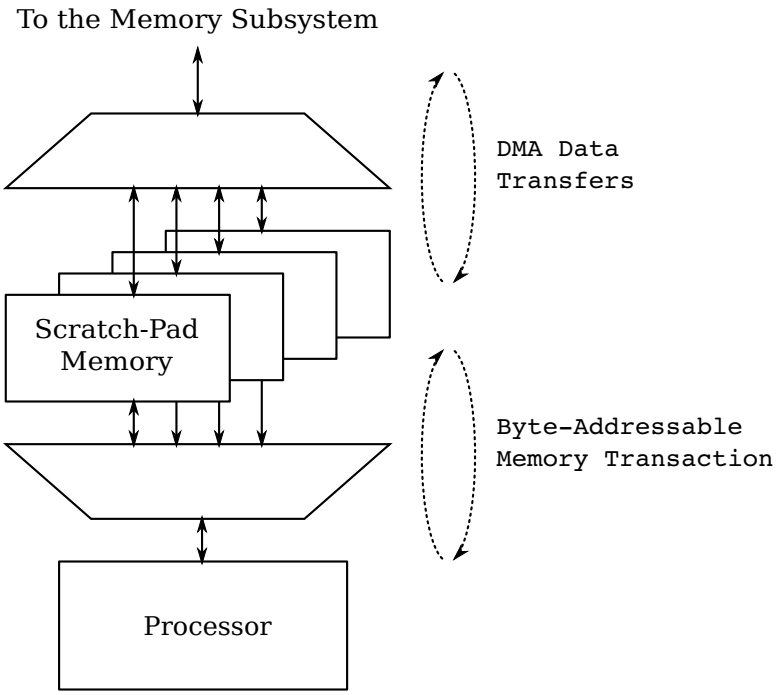


Figure 2.1: High level block diagram of a Green-Core

transfers. The central idea is to fit large number of parallel computations solely in the scratchpad, and making sure that enough memory transactions are available at any point to the subsystem such that it can efficiently use the available bandwidth. The data movement in green core is shown in the Figure 2.1.

This leads to the most important conclusion, that AMME is in fact, the workhorse for the green-core. The simplicity of the green core and the advantages it offers us, are only feasible if AMME can work fast enough to get the operands to GC. Hence AMME is designed with the perspective of speed and its implementation becomes important.

AMME is implemented with a B-tree model to store the node pointers and the metadata associated with each memory segment. The metadata can contain several useful fields such as key, left child, right child, its length and its status. All these entries are stored in a on-chip pool of BRAMs that is further managed by a B-tree controller that contains pointers to various locations in the pool. It has specific

registers for maintaining the root node. It has features to increase lookups and keep track of which node is in operation. It is also optimized to further know which node was most recently added by keeping it at a specific place in the BRAM table (at the top).

Each core when request a data from the AMME, it subscribes to that data. The master controller in the AMME design (also the controller that controls the on-chip pool stored in BRAM) keeps track of all data associated with the cores by adding and searching in the B-tree. If any data becomes redundant (old data after being written for example), then AMME can facilitate a delete operation using an epoch controller. All the reads and writes are done to the BRAM pool. Size of pool is resource dependent and also accounts for varied payload size that could be requested by the core. Detailed operation of a B-tree is described in the next section.

2.3 Concept of a B-tree

As described above, we are most concerned about the access time complexity of the AMME that is capable of handling varied sized payloads. As mentioned above, this is critical to the feasibility of AMME. The choice of a B-tree is further motivated by the reasoning because it is optimized for efficient use of bandwidth.

A B-tree is a generalized multi-way variant of a Binary Search Tree. In a binary search tree, at every node we make a two-way branching decision based on its key. If the desired key is lesser, we proceed to inspect the left subtree, or else we inspect the right subtree. In a B-tree however, instead of making a two way decision, we make a multiway decision to determine which child node to inspect. Just like a binary search tree, a B-tree is a sorted data structure. An example of a B-tree containing nodes and keys is given in Figure 2.2.

Each node of a B-tree can contain at the most a fixed number of keys. While different authors use slightly varied terminology to define how many keys a node can contain, we will use the terminology defined by Cormen [9] which is the Minimum

Degree. A B-tree has following properties that must be satisfied for it to be considered valid.

- Minimum degree poses a restriction on the minimum number of keys every node at any point must absolutely contain. Any node except the root containing less than minimum degree of keys, is a violation. If the minimum degree of a B-tree is t, then any node except the root at any point cannot have less than t-1 keys. Minimum degree is also used to define the upper bound on the number of keys. For a tree with minimum degree t, any node including the root cannot have more than 2t-1 keys. Hence what this also implies is a half-full condition. This means that any node except the root at any time is at least half full. The most common example is a 2-3-4 tree. In this case t is equal to 2. This means each node can have at least 2 keys and at most 4 keys (*2*t-1*). The minimum degree t has a logical condition that it has to be greater than 2. A minimum degree less than 2, will fail to be a valid B-tree. In practice however, we have much larger minimum degrees ranging in thousands.

- All the keys in each node are maintained in sorted order of increasing key values. This maintains the B-tree as a sorted data structure. In AMME, this feature is provided by a compare-sort module.

- Each node containing n keys with n greater or equal to t , must have n+1 children. This point is more logical than a property by itself. Since every key in a node can result in a branching decisions, for n keys in a node, we can have n+1 branching decisions. In addition to that, each node also maintains pointers to its children in the form of an array or a linked list, which are useful in recursing downwards.

- Unlike Binary Search Trees, all leaf nodes of a B-tree are at the same depth. This can also be referred to as the height of the tree. Height of the tree determines its

Table 2.1: Time complexity of various operations in a B-tree

Operation	Average Case	Worst Case
Space	*O(n)*	*O(n)*
Searching	*O(log n)*	*O(n)*
Insertion	*O(log n)*	*O(n)*
Deletion	*O(log n)*	*O(n)*

worst case running time, since height is essentially the number of levels we will have to recurse to get to the leaf node which may contain the desired object.

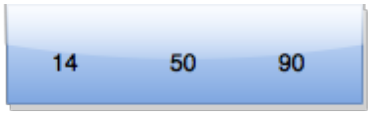
- A B-tree grows upwards, which again departs from the Binary Search Trees which grows downwards. If an insertion to the tree results in a full node, then the B-tree results in splitting of a node and a middle key value is selected as a parent node with its child pointers updated to reflect new children.

A search procedure in a B-tree recurses down the nodes from the root to the leaf. The theoretical time best case and worst case complexities of the B-tree structure are summarized in Table 2.1.

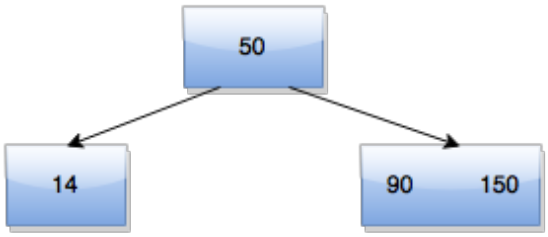
The number of memory accesses required by a B-tree in order to retrieve its keys are directly proportional to the height of the tree. To analyze the worst case height of a tree, we consider a tree with each node other than the root containing the minimum number of keys, and the root containing at least one key. This gives a relation between the number of keys in the tree and the worst case height of the tree. The worst case height h of the tree is given by Equation 2.1

$$h = \log_t \frac{n+1}{2} \tag{2.1}$$

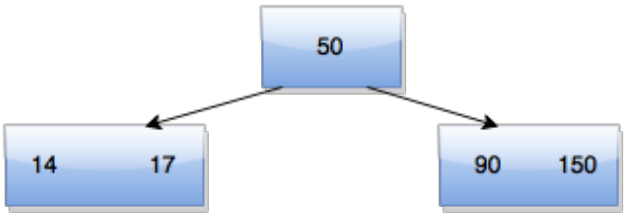
The best case height of a B-tree is calculated when we consider the case where all the nodes have the maximum number of keys possible. For a B-tree of degree t, the maximum number of keys it can contain is *2t-1*. Let m be the maximum number of children each node can contain. Therefore, the relation between m and t is given by



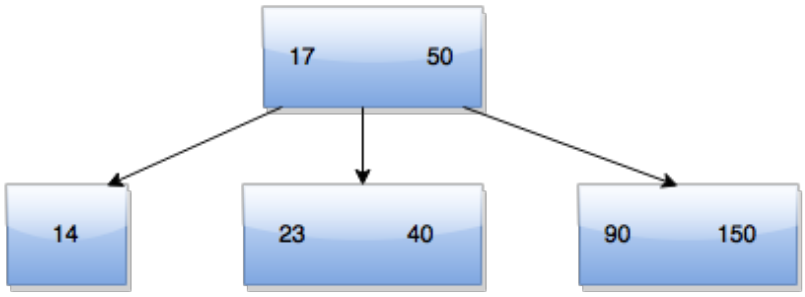
a) A root node containing 3 keys in sorted order



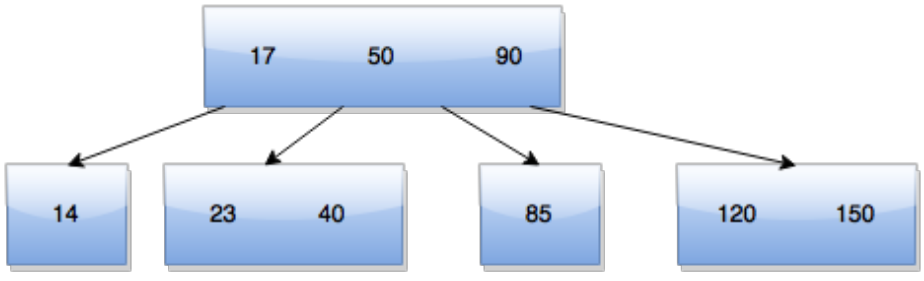
b) Inserting key 150 resulting in a split and root updated with pointers



c) Inserting key 17 in the left child node



d) Inserting key 23 in the left child node, and then key 40 resulting in splitting the left child node



e) Inserting key 85 and 120 resulting in a splitting and rearranging of the right node as each node can only contain at most 4 keys.

Figure 2.2: An example of insertions in a 2-3-4 B-tree with preemptive splits

Table 2.2: Best case heights of a B-tree with the given number of keys

Degree	500	5000	10,000	40,000
5	3	4	4	5
9	3	3	4	4
13	2	3	3	4
19	2	3	3	3
24	2	3	3	3

Table 2.3: Worst case heights of a B-tree with given number of keys

Degree	500	5000	10000	40000
5	5	7	7	8
9	3	5	5	6
13	3	4	4	5
19	3	3	4	4
24	2	3	3	4

Equation 2.2.

$$m = 2{*}t \tag{2.2}$$

The best case height h of a B-tree containing n keys is given by Equation 2.3.

$$h = \log_m (n + 1) \tag{2.3}$$

To give the reader an idea of how B-trees can maintain large number of keys and still maintain faster lookups, a table containing best case and worst case heights of a B-tree is given. The degrees chosen are the ones that are used throughout the experimental work done in this thesis. The size of the B-tree is determined by the number of keys chosen, which is taken from the datasets we have used, introduced in the next chapter.

Similarly, worst case heights are shown below in Table 2.3.

CHAPTER 3: DESIGN

The goal of this chapter is to describe the software process that was designed to test the memory controller implementation as a purely software dependent process. All the memory segment creation, reads and writes will be initiated by software itself. The idea is to test if software can in fact be comparable to the hardware implementation, if not better. Algorithms to do operations on the software B-tree such as create, insert, search and delete will be thoroughly discussed. Software B-tree uses various datasets, collected from real sources. Various other factors of the design such as the standalone processor, clock frequency, resource usage and various choices available at software level will be presented.

3.1 Hardware

Software based memory controller design setup is developed on a Ml-605 and Ml706 evaluation board. The tool chain used to synthesize various components of the design is Xilinx ISE version 14.5 for Virtex 6 and version 14.7 for Virtex 7. A basic Microblaze processor based system is chosen with external memory as DDR-SDRAM, BRAMS serving as caches as well as local memory to the processor, a memory mapped UART port set up to monitor the access times of the memory controller, and lastly a timer to count the clock cycles between various points of the search function of the B-tree.To understand the evaluation, its necessary to understand the data path that each instruction might take in all possible situations. The various components and how they come in the data path, are thoroughly explained.

3.1.1 MicroBlaze

MicroBlaze is a highly configurable, 32 bit, RISC based soft-core processor that can be optimized depending on requirement. It is frequently used for benchmarking as

well as testing various software prototypes. It can either be optimized for throughput or for area. The key difference between both the configurations are factors like- depth of pipeline i.e the pipeline latency and the way branches are handled. For throughput or speed based optimization, MicroBlaze can be configured as a 5 stage pipeline and for an area based optimization, MicroBlaze is configured as a 3 stage pipeline. Area optimization is typically most useful in situations where space constraints are too strict, and the system is low on resource. Various techniques used for speeding up the software such as advanced branch prediction, cache pre-fetching and the use of branch delay slots are avoided in area based optimization. Since speed is critical to our software prototype, we have used throughput based optimization. Micro blaze being a RISC machine, executes each instruction in at most two clock cycles. This means a processor running at a 100 Mhz, can almost execute 50 million instructions i.e 50 MIPS. Theoretically, performance of micro blaze is guaranteed at 100 MIPS [10]. All the data words are stored in Big Endian format. It uses memory mapped IO and hence does not differentiate between data access towards memory or IO.

3.1.2 Zynq Processing System

Zynq board contains the dual core ARM processing system and programmable logic fabric on a single board. The full fledged processing system is a system on a chip with its own L1 and L2 caches as well as its own memory management unit. It further has 256 Kilobytes of on chip memory as well as a memory interconnect to connect the application processing unit to the DDR memory controller. The on chip memory implemented on SRAM cells is at the same level as that of L2 cache and is hence not cacheable [11]. This memory has the lowest latency from the processor.

The ARM cortex A9 processor can be clocked at various frequencies. It has a flexible clock divider circuitry, that provides clocking to IO peripherals and the programmable fabric. This gives flexible clocking to not only the processor but also all the other functional blocks enabled in the system. For example for a base frequency of

33.33 Mhz, the frequency generated for ARM core (generated through PLL circuits) is referred to as ARM PLL and could be at a maximum of 800 Mhz. However DDR controller providing interface to DDR3-SDRAM is usually clocked with DDR PLL and runs at a frequency of 533Mhz. Similarly all IO peripherals use the IO PLL clocks that are clocked differently. The significance of understanding the clocking is important because if the instruction is fetched from the cache of the application processing unit, then its clocked at ARM PLL clock rate, and hence is fetched at a higher rate. However if an instruction takes a different data path and is clocked from DDR3-SDRAM then it will be fetched at a lower rate, which is added to the intrinsic latency associated with off chip access.

Zynq PS contains high performance AXI interconnects. The OCM interconnects connects to the central interconnect as well as 256 kilobytes of on chip SRAM. The central interconnect is 64 bits and connects to the DDR controller.

3.1.3 On-Chip Memory

MicroBlaze has on chip memory in the form of local memory RAM blocks connected directly via a bus. There are two LMB blocks, one for data (DLMB) and one for instruction (ILMB). This memory has zero data access latency from MicroBlaze. Any data residing on LMB will be accessed through the DLMB and ILMB ports directly connected to Mircoblaze and will take not more than 1 clock cycle (except for branches, floating point operations, division or pipeline hazards). This memory is configurable depending on application and is closest to the processor.

MicroBlaze core also have data and instruction caches implemented in the core itself which are used to cache the external memory. The caches use AXI4 interconnect to connect to the DDR-SDRAM. Although caches will reduce our latency and a 100 percent cache hit would be equivalent to storing data in the local memory block. However, because caching control circuitry is complex, overall performance with caches included in the system is usually no better than performance with the instructions

and data fetched from the local memory blocks. This is usually because some cycles get wasted in fetching cache lines when cache is empty, or evicting blocks on writes. Both caches and local memory blocks are however implemented in BRAMs. This aspect is studied in more detail in chapter 4.

While local memory blocks (DLMB and ILMB) are essentially BRAM blocks, it is not same as having additional BRAM added to the system. The DLMB and ILMB ports are separate that connect to the local memory blocks. If additional BRAM is added to the system, the memory accesses go through the AXI4 interconnect (explained below) and suffer a latency of 6-8 clock cycles. Thus contrary to intuition, simply adding more BRAM to the system does not improve the memory latency. The number of clock cycles expended will depend on where the memory is situated and how it is connected to the processor.

The buses used to connect the various components of the system also contribute to the latencies. AXI4Lite bus is typically used to connect to just peripherals which do not involve any time critical data movements. Its latency can be in the range of 6-8 clock cycles. AXI4 on the other hand is used to interface with the external memory, and can be used efficiently for memory transfers. The key difference is that AXI4 can support wider width data buses (up to 1024 bits wide) while AXI4Lite can only support at max 32 bits wide data. For single word line requests however, there is not much difference in latencies between AXI4 and AXI4Lite. Furthermore MicroBlaze uses two ports for AXI data access. The AXIDP ports is connected to the AXI4Lite interconnect which is used for peripherals (like UART) and AXIDC port which is connected to the AXI4 interconnect connecting the data cache inside the processor to the external memory. Micro blaze can do bursts requests on AXIDC port because thats the port used for fetching cache line requests. Increasing the width of AXIDC port to fetch more words on the cache line hence becomes a valid possibility to reduce latencies (evaluated in chapter 4).

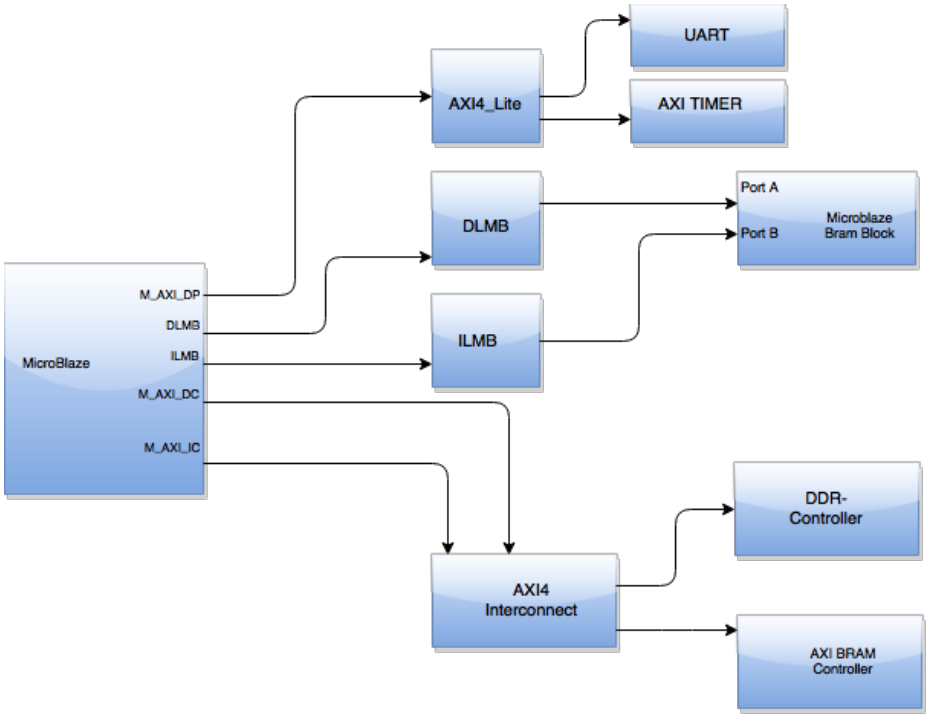


Figure 3.1: Hardware data path

3.1.4 External (Off-chip) Memory

External Memory is connected to the processor core via AXI4 interconnect and is cached in the local micro blaze memory, depending on the cache size initialized. In Ml-605 FPGA, we use DDR3-SDRAM as the external off chip memory initialized along with MIG controller. MIG can be configured to 64 bit data width. DDR3 operates at a frequency of 400 Mhz. DDR stands for double data rate meaning it can transfer data twice in a clock cycle, once on the rising edge and once on the falling edge. Off chip memory access, quite obviously is the slowest in terms of clock cycles.

Internal structure of DDR3 is organized as a two dimensional grid of rows containing word lines and columns containing bit lines. These grid like structures are called banks, which when associated in groups of 8, form the next logical unit called as a rank. External memory can have a latency ranging from 40 clock cycles mentioned in the data sheet to over a hundred clock cycles, based on the data access pattern. Controller can access the data on row-bank-column or bank-row-column addressing

scheme. The external memory controller typically tries to open channels to different banks to keep getting the data out continuously. Hence if rows in different banks are accessed then, controller can hide the latency. However, if the data access pattern is such that different rows are being accessed but in the same banks, then controller will fail to hide the latency and refresh cycles will only add to the total read time.

3.1.5 Peripherals

There are two peripherals included in the design. A UART peripheral connected over the AXI4_Lite bus is used to transfer the measurement data over the serial terminal. No interrupt operation is necessary for this peripheral, as any movement on this bus is independent of the data movement to the memory.

Similarly, a timer peripheral is connected to the AXI4Lite interconnect. AXI_timer is used as the timer peripheral, and again its used without any interrupt operation. Timer is used to measure the execution times of the code running from various parts of the memory. Since the reference clock is generated by the Microblaze core, the timer ticks with a frequency of 100 Mhz. To calculate the execution time of specific sections of the code, the timer is started at the start point of the section and its value is saved in a temporary register. At the end point of the section, the value of the timer is again noted. The difference in the two values, gives number of ticks during the execution of the code section. Since time interval for every tick is known, total time spent in that section can be found out.

```
Start Timer;
One_Tick_Start= Get_Timer_Value;
One_Tick_Stop=Get_Timer_Value;
Stop Timer;
Calibration=One_Tick_Stop - One_Tick_Start;
Start Timer;
   BeginTime= Get_Timer_Value;
```

```
                    // section of the code to be measured
    EndTime= Get_Timer_Value;
Stop Timer;
total_ticks=Endtime-BeginTime-Calibration;
```

3.2 Software Setup

B-tree software is designed as to specifically emulate the memory controller implementation. To design the data structure, a more object oriented approach was chosen. As described by the properties of the B-tree in chapter 2. To create a B-tree we create a root node that is capable of handling child pointers. In the AMME model [3], atomic unit of transfer between the controller and the BRAM pool is a node instead of just a key. Each nodes capabilities is defined with the minimum degree of the B-tree, which is configurable in software. Software maintains the pointer to the root node of the tree for traversal, which is similar to maintaining the root pointer in a specific location in the BRAM pool. The root node can also be cached in BRAM, for quicker access. For simplicity however, it is assumed that any satellite information is kept in the same node as the key itself [9].

Since creation of the tree (insertion of the keys) is done during the life of the program, the data structure is maintained on the heap. The placement of the heap can be done through the linker script generated. Implications of this will be discussed in the next section. For configuring the B-tree for different possible node sizes, the B-tree simply needs to be instantiated with the degree variable. This cannot however be done on the fly. To create a B-tree, we first instantiate a root node. Some important mnemonics are-

- Allocate-Node(x) : Every time a key needs to be added to the B-tree, the software performs a malloc to add the key to the data structure. Often existing node may be able to contain the key without needing additional space. But

because each node has a restriction on maximum keys, this step will allocate the new node and place it in the data structure.

- Memory-Read (x): Software would do a memory access to retrieve a particular key from the B-tree structure, and puts this value on the bus for the processor.

- Memory-Write(x) : Software would update the keys of the B-tree in case there is an addition to the node, or if the node is full and needs to be split. Memory-Write can also be performed on the root node. Due to insertions, often the root node will get changed. This is also because B-tree grows upwards and not downwards. Hence subsequent additions will change the root node. Since software only maintains the memory location of the root node, this does not make much difference.

3.2.1 Implementation

For adding keys into the B-tree T, we call a B-Tree-Insert procedure. This will require O(h) memory reads to recurse down the tree, to find the node where the new key can be added [9]. The insert procedure algorithm is defined in Algorithm 1.

Algorithm 1 Insert operation in a B-tree

```
B-Tree-Insert(T, key)
if root is empty then
    root= Allocate-Node (key);
end if
if root is full then
    node=Allocate-Node(key);
    select the middle key as new root;
    split the root node;
    insert key in one of the new child;
else
    if root is non-full then
        insert key in root;
    end if
end if
```

For inserting the key in the B-tree, we call many helper procedures like split-node

which would split a node if its full. Split node will take the median key of the current and split the remaining into two children and move it up, as the parent and all the child pointers of the parents would be updated to reflect the new children and the number of keys in parent would increase by one. However if the parent node is also full, then the splits go all the way till the time it finds a node with space to hold the key. Split node routine explained in Algorithm 2 dominates the time taken for inserts in the B-tree data structure.

Algorithm 2 Algorithm showing splitting of a full child of a node

```
SplitChild(Node,y)                          ▷ Splitting the child y of node Node
Allocate-Node(x)                                          ▷ Holds the new child
Copy t-1 keys of y to x
Copy t children of y to x
Update no. of keys in y
Create Space in Node to hold new child x
Add a child pointer in Node pointing to x
Move median key of y to Node
Increment number of keys in Node
```

The search function of the B-tree recursively traverses the tree until it finds the requested key. Since all the keys in the node are kept in the sorted order, it simply checks each key if its greater than the requested key, and if yes, then it traverses down the child pointers. The Search algorithm is defined in Algorithm 3.

Algorithm 3 Search operation in a B-tree

```
B-Tree-Search(node, key)
for all keys in the node
if node.key== key then
    return the node.key;
end if
if node is a leaf then
    return not found;
else
    Memory-Read( node.child)
    B-Tree-Search(child, key);
end if
```

3.2.2 Compiler and Linker

Xilinx Software Development Kit is used for all software simulations. The C++ implementation is cross compiled with GCC compiler for MicroBlaze and finally linked into an ELF file to be loaded on the board. The compiler with no optimization produces a direct translation of the written code which often contains more instructions than it absolutely needs consuming lot more space and clock cycles than necessary. For this reason, its necessary to pay attention to optimization. Our fundamental goal here is to produce a lightweight, highly optimized code that can fit into different sections of the memory. Speed of the code is also essential to our application. Since we are running a 5-stage pipeline with full features of micro blaze such as branch prediction and instruction prefetching, the code needs to efficiently take advantage of the full hardware support offered my MicroBlaze.

Some important hardware features that speed up the code are use of barrel shifters, and hardware multipliers for quick computations. Since there is special hardware for these instructions or instructions that translate to shifting and multiplication (in loop counters etc), these instructions can execute in exactly one clock cycle and do not stall the pipeline. To enable the support for such instructions, the code needs to be compiled with special flags that tell the compiler that the available hardware support will be used.

Various other compiler level techniques that help to speed up the code, such as loop unrolling for long loops containing a set of non-branch instructions, register renaming etc can be used with GCC optimization. We have used -Os optimization for size which apart from reducing the size of the code, also unrolls loops and reorders instructions that can lead to more efficient code.

The linker script for the application controls where each of the code sections are going to be placed. Since the code uses a lot of malloc and free calls, the structure resides on the heap memory of the system and cached in the BRAM memory. The

stack and heap grow in opposite directions and as such cannot be assigned to different memories. If the application is run on DDR3- SDRAM, then heap and stack both need to be assigned to DDR3. For measuring the latencies from BRAMs, the heap and stack can be assigned to BRAM. Since really large datasets are used, the size of heap grows and this can be a limiting factor especially if BRAMs are limited. The code also uses a lot of recursive calls, and hence stack usage can be high especially in the search function.

3.2.3 Datasets

Datasets used for the software implementation are used from real life data to understand and design the system with real life patterns that frequently occur in the data. The first data set contains encrypted hashed outputs of Amazon food reviews [12] taken from a hashing setup implemented using SHA-3 core [13]. It contains up to 11111 hashed keys that are stored on the software simulated B-tree structure. The second data set considered is taken form the server data containing absolute file pathnames to over a million files and directories. This data set contains hashed keys for the file pathname data and contains over a 40,000 keys.

CHAPTER 4: EVALUATION

Performance of any application be a tricky thing to quantify. For a software memory controller, the two most important things that can characterize its performance are memory latency and resource utilization. For an ideal memory controller resource utilization and latency should be as low as possible. In our application, resource utilization on an FPGA can be characterized by number of LUT's used as well as a number BRAM blocks utilized. To measure the latency, we measure the access times that represent the time it takes for the controller to search any given key in the B-tree starting from the root node. We also measure the time it takes to do insert operations in a B-tree. These access times are critical to the operation of the memory controller since it provides the upper limit as to how fast the controller can retrieve keys.

This chapter is organized as follows. The Section 4.1 deals with the base system set up, and shows the results obtained in case of lookups and inserts for two datasets, to show any co-dependeny between the nature of dataset.There is a subsection 4.1.1 that deals with the results differing as the result of the size of the dataset. In section 4.2 deals with some modifications applied to the results of Section 4.1. Section 4.3 deals with the more modifications applied to results of Section 4.1 and improvements studied. Finally we present a final evaluation summarised in Section 4.4 that combines the factors studied in both the prior sections. There was no necessity observed for a dataset variation study for this set of results.

4.1 Initial Evaluation

For the initial experiment, a base system containing a Microblaze processor was setup as described in chapter 3. The goal of this experiment is to measure the access times of the B-tree based controller when implemented purely in software. The

Table 4.1: Lookup access times using the Amazon data set

Degree	Average case	Worst case	Best case	Median
5	3.34	6.57	0.05	3.36
9	3.47	7.73	0.05	3.44
13	4.50	9.89	0.05	4.48
19	4.20	11.87	0.05	5.13
24	5.43	12.65	0.05	5.40

application executes from off chip DDR3 memory. The latency of external memory is the highest, but this experiment was setup to get a rough estimate of exactly how much clock cycles are we spending in accessing this memory. The system is designed with some on chip internal memory of the Microblaze, implemented through BRAM blocks. Microblaze also has additional on chip cache memory set up. Since both the on-chip memory as well as cache memory is implemented in BRAM, it is tricky to think of obtaining a speedup using both on chip-memories. This fact is counterintuitive since both the memories are at an equal distance from the processor, the caching overheads actually increase the latency of cache memory. Even though in most cases caches are simply assumed to provide a speedup but cache circuitry is still complex. There are cycles expended in cold cache misses, flushing cache lines, evicting victim blocks and conflict misses. However such complexity is still tolerable when we are trying to hide the latency of the main memory which uses hundreds of clock cycles for each access. But in the case of local on chip memory of the Microblaze processor, the caching does not provide any further reduction in clock cycles. On the contrary, caching may become costly. In this experiment we enable 64 Kb data and instruction caches to hide as much off chip latency as possible.

The results of the amazon data set are presented in the Table 4.1 showing the various access times when the B-tree Controller runs from the external memory. The Axi_Timer core is clocked at 100 Mhz which yields a time period of 10 ns. A plot showing average, worst-case and best case access times are presented in Figure 4.1.

A second data set containing hashed values representing absolute paths of over

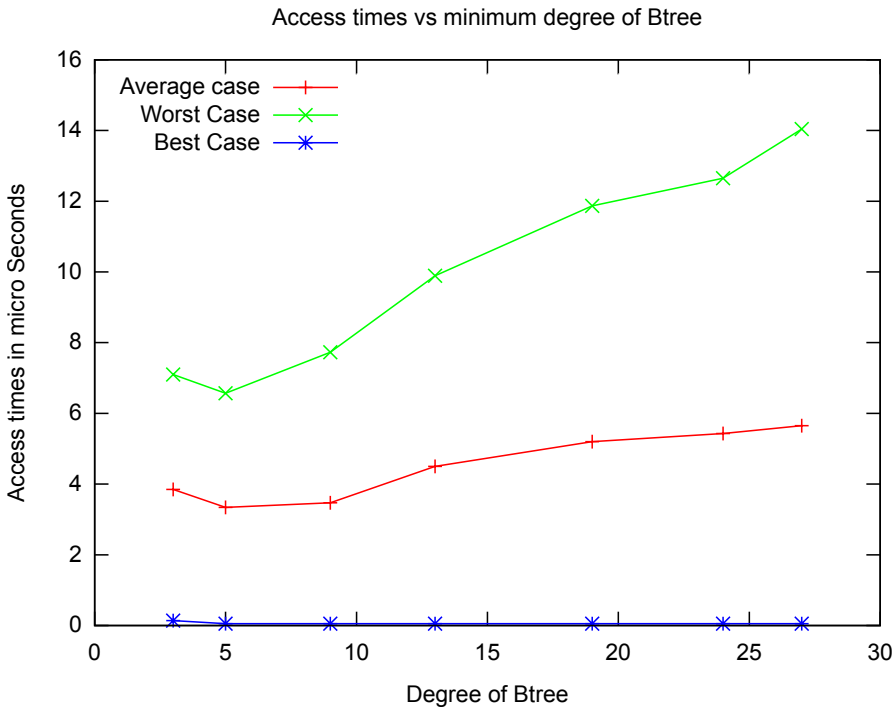


Figure 4.1: Lookup access times of the B-tree using Amazon data

Table 4.2: Lookup access times using server data set

Degree	Average case	Worst case	Best case	Median
5	4.51	9.21	0.05	4.58
13	5.34	11.57	0.14	5.31
19	6.93	15.66	0.05	5.32
24	7.79	18.24	0.05	7.74
27	7.67	18.10	0.05	7.59

a million filenames collected from a server is stored in the off-chip memory. Access times from this data sets are shown in the Table 4.2below and plotted in Figure 4.2.

We also measure the time it takes to insert a key in the B-tree. This metric is useful to understand the best case and worst case estimates. Some inserts in the best case, will simply be in the root node. Average case inserts will be in a middle node or even in a leaf node but will just require some traversal down the tree. However worst case inserts happen when a key needs to be added to a node that is already full and needs to be split. Since the median key of the node moves to the parent, if

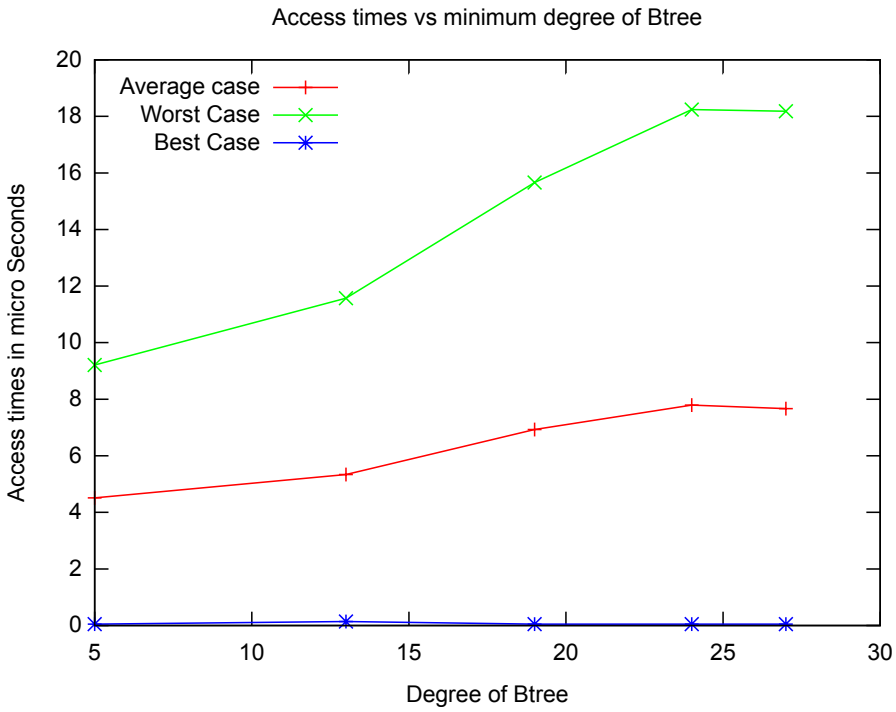


Figure 4.2: Lookup access times using server data set

Table 4.3: Time to insert a key from Amazon dataset

Degree	Average Case	Best Case	Worst Case	Median
5	5.71	0.80	35.08	3.82
9	5.14	0.48	32.81	4.01
13	5.46	0.49	35.44	4.60
19	5.95	0.64	36.21	5.32
24	6.49	0.71	34.75	5.90

the parent is also full, the nodes above keep on splitting until the root node. This constitutes the absolute worst-case.

The results in Table 4.3 show the access times to insert 10,000 keys in a B-tree of different orders.

We are most interested in the average case and the median access times in lookups and inserts, since those access times will dominate the overall latency of the application. These access times are quite high compared to the hardware based controller. A quick comparison shows, that this performance is intolerable for any controller to be

implemented in software since speed is critical. This leaves us with two fundamental questions namely- What factors contribute to such high access time (latency) and secondly, what factor (or combination of factors) if changed, can result in better performance?

To answer the first question, we examine the data path explained in chapter 3. As Figure 3.1 shows, off chip memory is cached in the on chip Bram cache blocks of Microblaze. Due to architectural limitations of Microblaze cache size can only be increased up till 64 Kb for both data cache and instruction cache. The distance of the off chip memory from the processor plays a major role in the latency since for every instruction fetch to an address in the DDR3-SDRAM, we spend at least 70-100 clock cycles also taking into account the latency of the AXI interconnect that accounts for 2-3 clock cycles. After an instruction is fetched, the data fetches especially when the cache is cold, account for another 70-90 clock cycles. For a B-tree based controller, the root node only needs to be fetched once when the cache is cold. For all subsequent accesses, since root node is in the cache, we spend less cycles in the data fetch. Depending on the nature of accesses, there might be some thrashing of cache involved. However all accesses are made completely random to mimic a worst-case scenario. The software function to search and access the keys of the B-tree comprises of 30 instructions out of which one thirds are load word instructions and another third being branches. With each load we pay the off-chip penalty mentioned above and with each branch we flush the pipeline resulting in additional clock cycles spent in cache accesses or in worst case another off chip access.

We are specifically more interested in knowing the percentage of execution time spent in various pieces of the function. Out of these, we are most concerned about the loads and branches, since we pay highest penalty here. Software intrusive profiling is a good way to find out the time spent in various functions. However it is not as specific as to tell us percentage time per instruction. To measure percentage time

Table 4.4: Percentage of execution time

Type of instruction	Percentage of execution time
Loads	68.1%
Branches(Approximate)	23.3%
Arithmatic operations, data movement	8.6%

Table 4.5: Resource utilization

Slice Registers	5%
Slice LUT's	8%
No. of Occupied Slices	10%

per instruction we can simply write a simple assembly code to analyze loads and time it using the hardware timer. Since timing of each instruction alone cannot be deterministic because of pipelining and instructions before and after it, therefore the instruction is executed 100 times for loading different values from off-chip memory (array like accesses), and then an average time is taken. Similarly average time taken for arithmetic instructions can be found out. This method however cannot be used to deterministically understand branch behavior because branch behavior is dependent on the current state of the program and the cache which cannot be simulated outside. To understand branch penalty we can only make an approximate guess.

In our search function, we already know the access times for finding a key in the B-tree, and we also know that one third instructions are loads and another third are branches, we can easily calculate the percentage time spent in loads and branches. This does take into account the time saving we get through caches because the assembly code also runs with cache enabled. The results of these calculations are presented in Table 4.4.

We are also concerned with resource utilization of the application on FPGA. This application is developed on ML-605 (XC6VLX240T) which contains 301,440 slice registers and 150,720 LUT's[10]. Since we are running the whole application from the DDR3-SDRAM, our BRAM utilization is low. The resource utilization for this application is shown in Table 4.5.

These results show that maximum penalty is attributed to off-chip latency. This leads us to our second question- What factors or combination of factors can help us improve the performance of software memory controller? In short, can we do better? To address this, we will first try reducing the latency by keeping most accesses (if not all) on chip. This is explained in detail in Section 4.2.

4.1.1 Dataset Variation

As seen above, there is some variation between different sizes of datasets. The results shown with two different datasets which are obtained from widely different sources, show that variation in nature of dataset has less or no impact on the access time patterns. However size of the dataset has a role to play, which is also intuitively correct. The size of the data set determines the number of keys the B-tree will contain and hence the depth of the tree for various orders. Since depth of the tree will decide the number of hops required to access any particular node (especially a leaf node). Hence as quite expected, as the size of the B-tree grows, the average and worst case access times increase. The best case however remains more or less the same (with very minor difference).

To clearly understand the variation in average access times corresponding to the size of the datasets w.r.t Amazon dataset containing 10,000 keys is presented in the Table 4.6. All values are taken for different orders of the B-tree given in column 1. Column 2 gives the average case access times for the data set containing 500 keys, and column 3 provides by how much percentage it varies from the Amazon Data set. Similarly column 3 and 4 provide average access times for dataset containing 5000 keys and the variation with amazon dataset respectively. Similarly to understand data set variation in case of inserts, the Table 4.7 shows the variation observed across various dataset sizes. Since the size of the data set would govern the height of the tree, we do see the difference in the worst case height of the tree.

Table 4.6: Variation in average lookup time (microseconds) with dataset size

Degree	500 keys	%Variation	5000 Keys	%Variation
5	1.90	37.36%	3.17	0.05%
9	2.28	34.29%	3.58	3%
13	2.62	41.77%	3.94	12.4 %
19	3.17	24.5%	4.38	4.2%
24	3.33	38.67%	5.45	0.003%

Table 4.7: Variation in average time to insert 500 and 5000 keys

Degree	500 Keys	% Variation	5000 Keys	% Variation
5	4.43	22.41%	5.18	9.2%
9	3.74	27,23%	4.58	10.89%
13	3.85	29.48%	4.71	13.7%
19	4.04	32.10%	5.25	11.7%
24	4.33	33.38%	5.59	13.86%

4.2 Improving Latency

The goal of this experiment is as stated above- keep all accesses close to the processor, i.e. on chip only. To do this, we will attempt to keep the B-tree structure containing all the keys mainly in the local on chip memory. Since the data structure will have most accesses, this achieves our goal. For this section we modify the linking of the software application. The linker script controls how the various sections are arranged in memory. Since size of on chip memory is limited, we can only make limited improvements here. But this answers the question, of whether improving latency alone can be beneficial enough? The B-tree is created dynamically in the software, so it resides in the heap area. To control the placement of heap, we are forced to move the stack too (Heap and stack can only be put together). Maximum on-chip memory possible is 128 Kilobytes. This is not enough to hold larger data set such as the Amazon's data set used earlier. For this we use a smaller subset of Amazon data that can conveniently reside on the 128 Kb memory. Since B-tree search function is recursive, we require a heavy stack usage. The stack also limits the size of b-tree we can maintain on-chip. For this application, at least 10 kilobytes worth

Table 4.8: Lookup access times using(500 keys) on-chip memory

Degree	Average case	Worst case	Best case	Median	Average case improvement
5	1.19	4.74	0.07	1.28	37.36%
9	1.52	4.22	0.07	1.30	33.33%
13	2.00	5.11	0.07	1.69	23.6%
19	2.48	6.35	0.07	2.08	27.82%
24	2.43	6.35	0.07	2.86	27.02%

of stack is necessary (-O3 optimization further increases stack usage). As already mentioned, for on-chip accesses, caches dont play much important role, as our data is already as close to the processor as it possibly can be. In fact, caching overheads (cold cache, misses, flushing) can only worsen the situation. Even though we dont need cache, but since other sections of our applications, such as text section, data section etc are still residing on DDR3 (resource limitation), we are forced to include caches.

To reduce further dependence on external memory, another BRAM block was added. However, this memory is not technically on-chip. The additional BRAM block is still connected through AXI interconnect. This implies it will still suffer a latency of 8-10 clock cycles (All data accesses over AXI suffer a latency of at least 2-3 clock cycles). However this is still better than DDR3-SDRAM. A 128 Kb additional BRAM block is included to move all other sections of code (except the .text) away from external memory. The data set used contains 500 keys. The dataset variation was already mentioned in Table 4.6. The average case improvement is calculated as the improvement seen in the average case access time when the B-tree is maintained off-chip completely with the 500 keys dataset (as shown in Table 4.6 versus the B-tree maintained on-chip, as well as some pieces of the code kept closer to the processor. The results are shown in Table 4.8.

For this experiment resource utilization numbers are given in Table 4.9. Our BRAM utilization is higher than before as major sections of the code are maintained on chip.

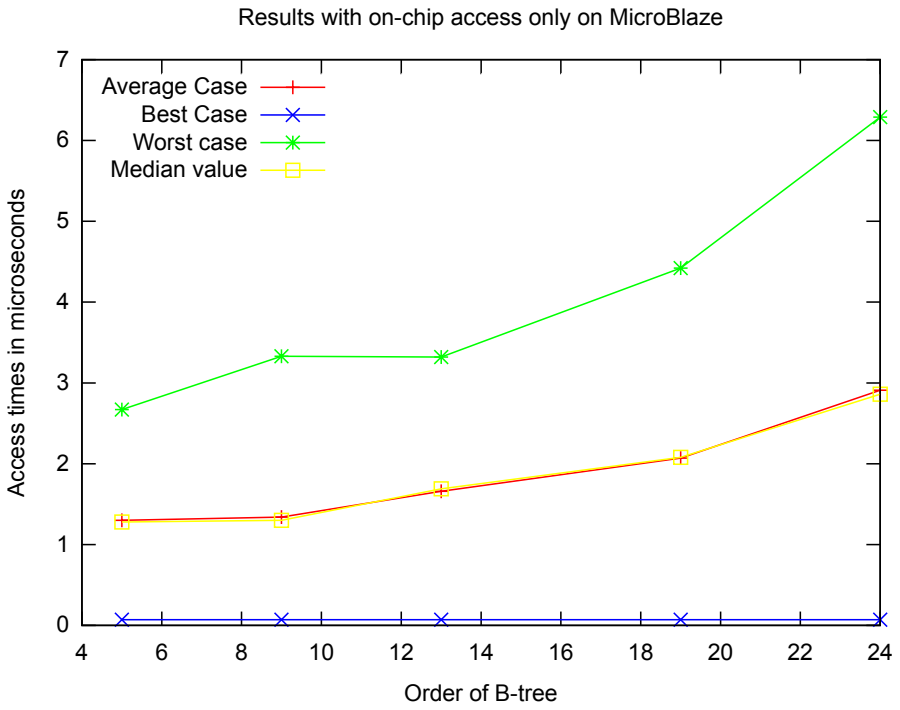


Figure 4.3: Lookup access times of on-chip B-tree

Table 4.9: Resource utilization with most accesses restrained to on-chip

Slice registers	7%
Slice LUT's	10 %
No. of Occupied Slices	20%

Table 4.10: Time(microseconds) to insert keys in on-chip B-tree

Degree	Average Case	Best Case	Worst Case	Median	Average case improvement
5	4.11	0.55	27.95	1.75	7.2%
9	3.13	0.42	28.31	1.51	16.31%
13	3.84	0.41	31.69	3.26	0.002%
19	3.54	0.41	29.76	2.89	12.3%
24	3.72	0.41	31.71	2.74	14.08%

For B-tree maintained off-chip, times for inserts in a B-tree of dataset 500 is also given in Table 4.10. Average Case improvement is considered from access times obtained in Table 4.7. The speedup offered over the previous implementation is shown in Figure 4.3. Even after this, the performance of the memory controller is still not comparable to hardware based design. The latency of the off-chip memory is not completely hidden due to some parts of application still residing on it. That means not all memory accesses are restrained to on-chip because of not enough on-chip memory is available. Since improving memory latency has resulted in an improvement of over 25%, the important question is, if the speed of the processor is improved can that make things better? Since we are right now still not dealing with branch penalties, slow clocking speeds, and overall slower architecture and the delays resulting in the system because of that. To make sure the delays are not compute bound, the question answered by the next set experiments is, can we still do better?

4.3 Improving Speed

The main goal of this set of experiments is to increase the overall speed of execution and to see how it impacts the performance. The idea is to eliminate any compute bound delays, and determine whether the performance obtained can actually make software based design a viable solution or not. For this purpose, we turn towards ARM cores. We use the Zynq board which contains a dual-core ARM based full fledged processing system. It has a complete System on a Chip which has certain customizable features, as explained in detail in chapter 3. We use the programmable logic fabric as well to synthesize an AXI timer core, for measuring the execution times.

Table 4.11: Lookup times(nanoseconds) on ARM with off-chip B-tree

Degree	Average case	Worst case	Best case	Median	Average case improvement
5	449.1	1020	7.5	442.5	86.5%
9	487.5	907.5	15	480	85.97%
13	451.87	937.5	7.5	457.5	89.95%
19	443.25	1072.5	7.5	442.5	91.47%
24	456.31	952.5	7.5	457.5	91.59%

Table 4.12: Time (ns) to insert in off-chip B-tree on Zynq PS

Degree	Average Case	Best Case	Worst Case	Median	Average Case Improvement
5	491.25	7.5	2460	420	91.3%
9	414.79	20	3540	340	91.93%
13	396	20	3320	340	92.7%
19	363.6	20	3540	340	93.89%
24	370	20	3380	360	94.20%

For the first set of results, the application is executed from DDR3-SDRAM. It has one gigabyte of memory. Application executing from this memory will suffer from memory latency delays that we saw in the first set of experiments. However to measure the performance gain from using a faster processor in isolation, we will use the off-chip memory. The results from the ARM core are presented in Table 4.11. Average case improvement is considered as the improvement obtained in average case access time from Table 4.1 versus running it on ARM SoC from DDR3 memory.The ARM core is clocked at 667 Mhz and the timer connected to arm clocks and clock divider circuits, is clocked at 133.33 MHz resulting in a time period of each tick equivalent to 7.5 ns.

The times to insert 10,000 keys in the B-tree is shown in Table 4.12. Average case improvement is considered the percentage improvement gained from Table 4.3.

Results on ARM core show significant improvement over results on Microblaze. Since ARM is clocked at more than 6 times the frequency, we can naturally expect at least thrice as much speedup, which is evident from Figure 4.4. ARM cores also have 32 kilobytes of data and instruction L1 caches and 512 KiloBytes of L2 cache. With these figures we are clearly very close to the hardware equivalent B-tree based con-

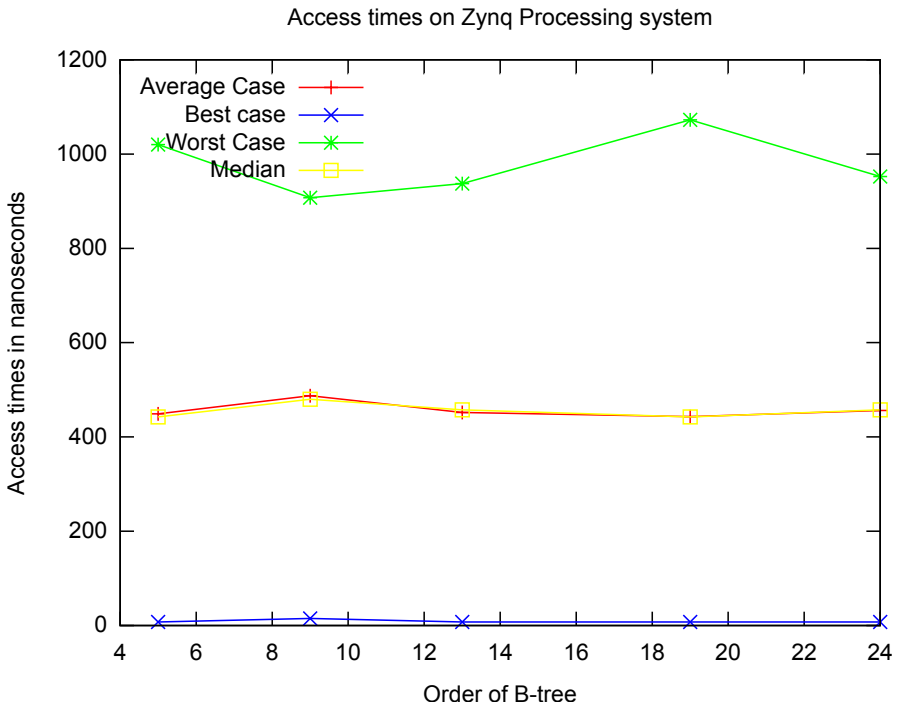


Figure 4.4: Lookup times on ARM with off-chip B-tree

troller. The on-chip resource utilization is not as high in the case of Zynq processing system. Any compute bound factors that result in slow access times are eliminated in the Zynq PS. However we still face memory bound delays.

The Zynq board is a -2 speed grade which means it can be clocked at a maximum of 800Mhz. The next set of results show if clocking at 800Mhz provides any significant speedup.The goal of increasing the clock speed is to make sure all compute bound factors are taken care of and we are absolutely sure that any remaining factors are not due to processing capabilities. The timer is clocked at 200 Mhz which results in a time period of 5 ns. The dataset used is Amazon dataset containing 10,000 keys. The results for this are summarized in Table 4.13. The average case improvement factor in these results is taken as the improvement obtained from running the ARM core at 667 Mhz versus running the ARM core at 800 Mhz.

As seen from the results, we do not see a high performance improvement from increasing just the clock rate of the ARM processor. The results obtained for inserts

Table 4.13: Lookup times (ns) on ARM running at 800Mhz with off-chip B-tree

Degree	Average case	Worst case	Best case	Median	Average case improvement
5	410	780	20	415	8.7%
9	344.5	680	10	345	29.3%
13	413	805	5	435	8.6%
19	436.75	875	15	440	1.4%
24	452.51	940	5	450	0.9%

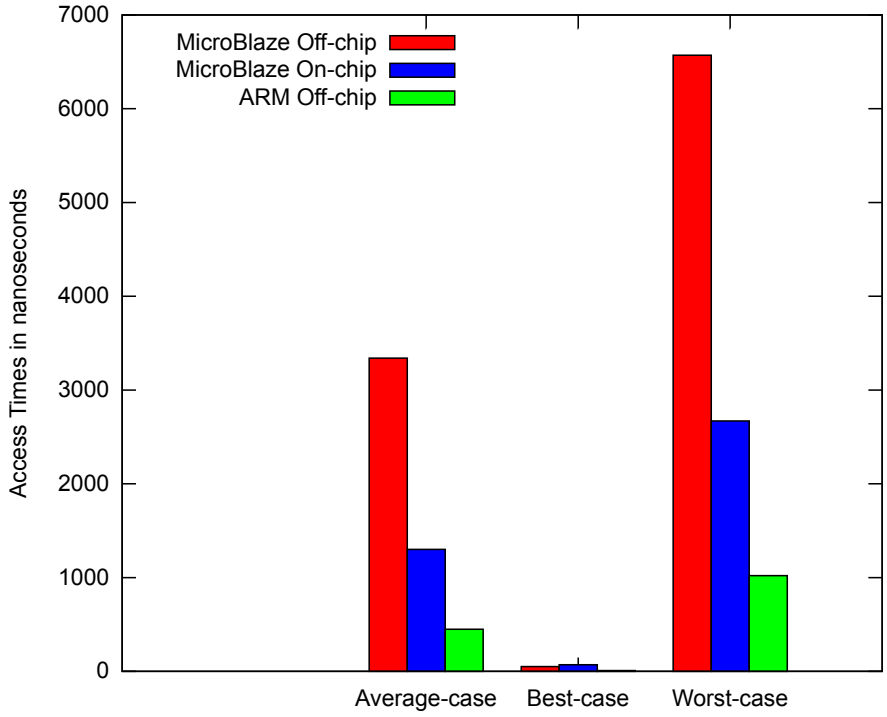


Figure 4.5: Improvement due to hiding latency and increasing speed

do not show a drastic improvement either. This means we are facing memory bound delays. Since the application is run from the DDR3 memory, we are still far from the processor in terms of the data path and the latency to access DDR3. All latencies over AXI interconnect have at least a latency of 10-20 clock cycles. This also implies that we have achieved the limit when it comes to processing related speedup.

To summarize the various results seen so far, the graph in Figure 4.5 shows the comparison of lookup access times and the speedup obtained for an order-5 B-tree.

Table 4.14: Lookup times (ns) on ARM at 800Mhz and on-chip B-tree

Degree	Average Case	Worst Case	Best Case	Median	Average case improvement
5	140.95	285	30	145	68%
9	132.05	320	25	130	72.81%
13	147.8	320	15	140	67.2%
19	165.9	365	25	160	62.5%
24	216	310	30	210	52.6%

Table 4.15: Time to insert in on-chip B-tree on ARM core at 800 Mhz

Degree	Average Case	Best Case	Worst Case	Median	Average Case Improvement
5	290.7	25	2050	200	40.90%
9	247.3	25	1935	200	40.37%
13	251	50	2045	200	36.6%
19	232.5	25	2080	200	36.05%
24	236.9	25	2035	200	35.97%

4.4 Final Evaluation

Finally we have seen that increasing speed of the processor significantly impacts the performance, offering improvements over 85%. Increasing clock speed from 667 Mhz to 800Mhz offers only minor improvements. We have also seen that latency can be improved by over 25% by keeping most accesses close to the processor. The purpose of this set of experiments is to determine the combined impact of the performance of the system with both the factors taken together. The Average case improvement is calculated by taking the average access time when ARM core is running at 667 Mhz with off-chip accesses (DDR3-SDRAM accesses) shown in Table 4.11 versus average access time obtained with ARM running on 800 Mhz but most data accesses to the B-tree structure kept on-chip (with the size of the B-tree in terms of number of keys kept constant).

Similarly insert access times are given in Table 4.15. The number of keys inserted is 10,000 (commensurate with the size of the B-tree that the on-chip memory can hold). Average Case improvement is considered as the percentage reduction obtained compared to results from Table 4.12

These results finally tell us that overall improvement in average case as a com-

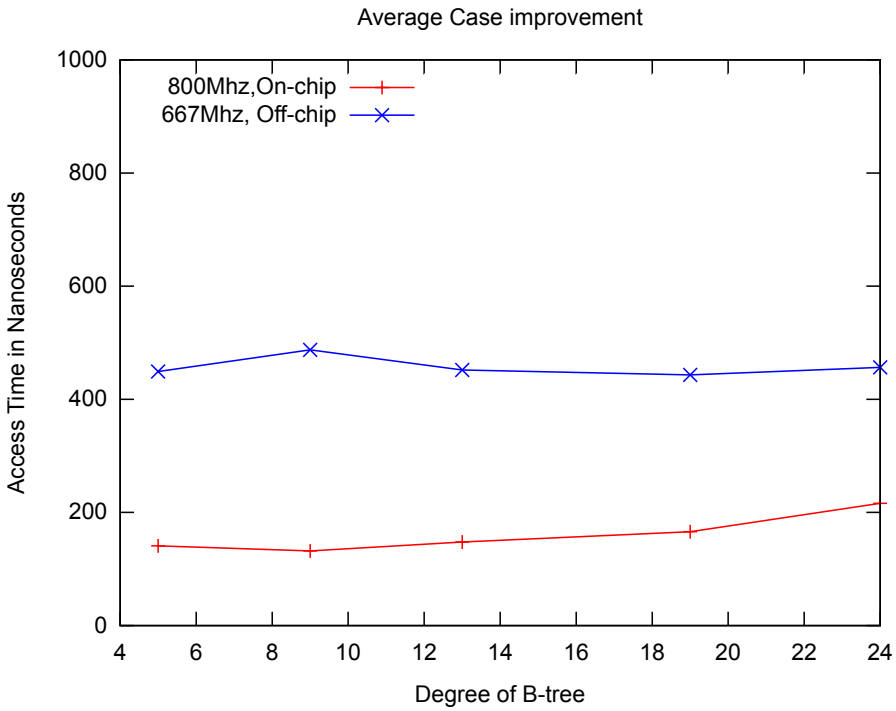


Figure 4.6: Comparing lookup times in Table 4.11 and Table 4.14

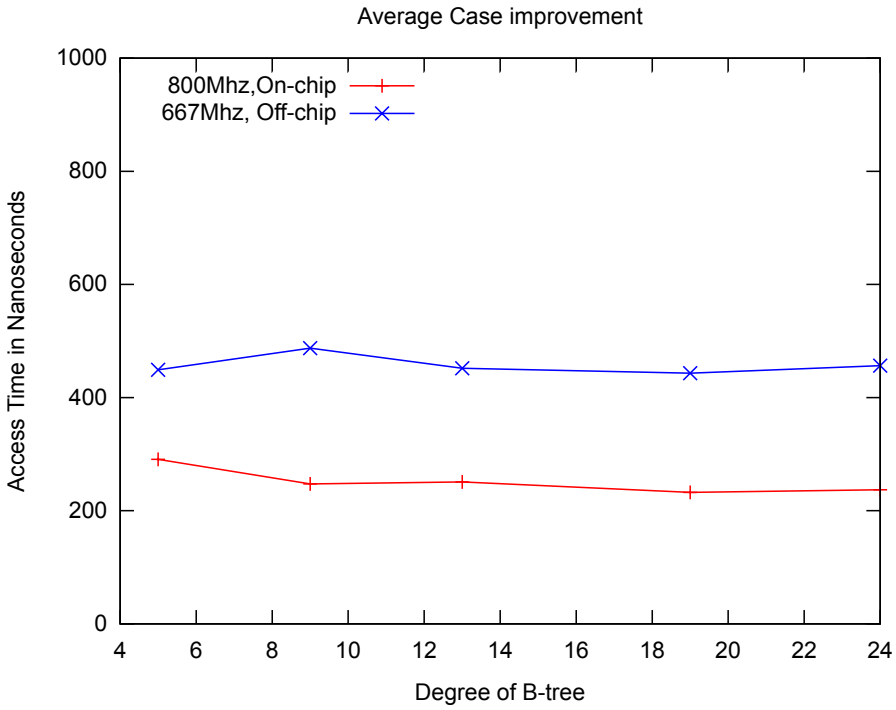


Figure 4.7: Comparing insert times in Table 4.12 and Table 4.15

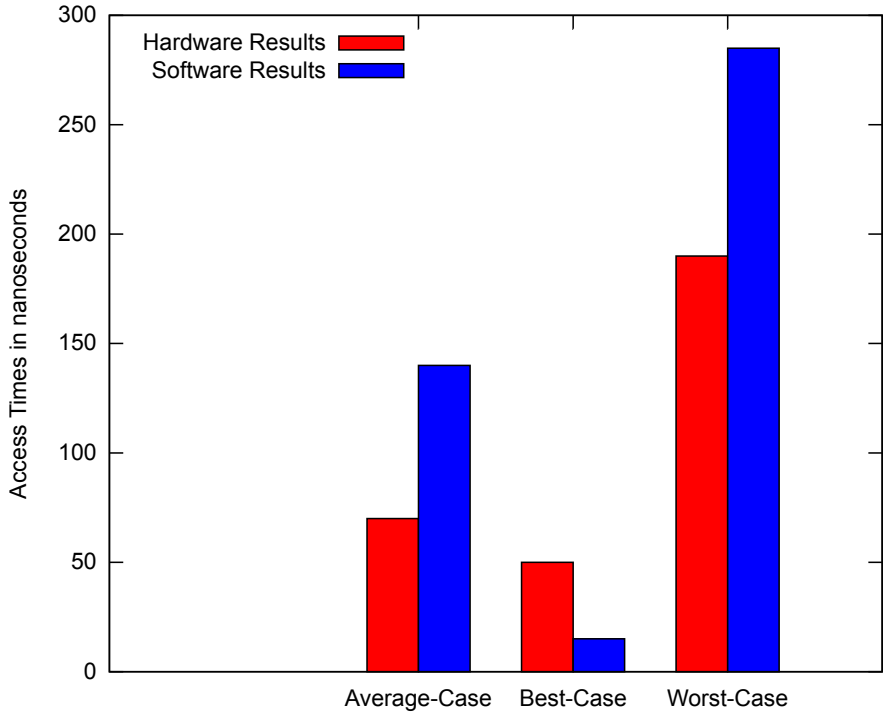


Figure 4.8: Comparison of B-tree access times on hardware vs software

Table 4.16: Resource utilization on Zynq-PS

Slice Register	1%
Slice LUT's	8%
No. Of occupied slices	1%

bination of two factors is more than 50% which is significant considering that this measurement is taken from ARM experiments from DDR3-SDRAM. Since improving speed itself offered an improvement of over 85%, we can assume that the speedup obtained from the Microblaze processor is over 200%. This gives us hope that a fast processor combined with low latency memory can speedup the operation of the memory controller.

Finally, the resource utilization on Zynq-PS is shown below. A comparison with results obtained from the hardware implementation of a B-tree is shown in the Figure 4.8. Also the resource utilization numbers are presented for the Zynq PS in Table 4.16. Some of the key observations from this experiment are-

- The best case access time, is actually much lesser than the best case access time obtained in hardware. This is a positive result. To understand the best case better, its necessary to know when it occurs exactly. The best case for a lookup occurs when the desired key is either in the root node itself, or at the least, one hop away from the root node. Not only that, a cold cache miss will result in the root node (and subsequently the child nodes depending on the cache line size) to be fetched in the cache. Therefore all subsequent accesses to either a key in the same node or nearby node will result in the best case scenario. On a generic level, all cases of cache hits will result in a best case scenario. The degree of the B-tree will also affect the best case condition.

- To understand this, we recall the definition of minimum degree of a B-tree. The minimum degree puts a restriction on the minimum and maximum number of keys a node can contain. While the search time in a B-tree is logarithmic w.r.t height of the tree, the search time within a node is linear. Since each node is essentially sorted with increasing order of keys this time can be made logarithmic as well. However the key thing is, as the degree increases, the time to search within a node will only increase.

- The worst case condition will happen, when the key requested is within a leaf node. This will require more hops from the root node and traversals of all nodes in between. Moreover, if this leaf node or any other node preceding it is in the external off-chip memory then it will only become worse. Thus a key residing in the leaf node which is uncached and requires one or more off-chip accesses contributes to the absolute worst case

- From analyzing the results of the final evaluation, we see two key observations. Firstly, the median value is close to the mean. This implies that the average case is also the most recurrent case. Secondly, the upper quartile is close to

the mean. Upper quartile divides the top 25% values, and refers to the 75th percentile. What this means is that 75% values are actually close to the mean value and top 25% are spread between the average and the worst case access time. Since the distribution is skewed, we can conclude that average case is dominant.

CHAPTER 5: CONCLUSION

Understanding the design space involves taking into account various factors that can impact the performance of the system. Its important to understand which decisions have the most impact and how changing one decision can impact the others.

We started with an intuitive assumption that for a given hardware implementation of a B-tree Controller, a similar software implementation is guaranteed to be slower and much less inefficient, rendering it as unfeasible. The core motive then was to understand why, it is and what factors contribute. We then went on to isolate some factors and study the improvement they offered when applied to our software setup. similarly we finally implemented a complete setup containing modifications to improve various factors, and eventually arrived at results that are much more comparable to hardware implementation. Thus to answer the thesis question, yes we can in fact change more than one factor such as off-chip latency, clock speeds, faster application processing unit. Combining and modifying these can in fact close the gap between hardware and software implementation between the B-tree Controller.

The results we have obtained show us that software can be very comptetive in a stronger but generic hardware background. The question this thesis raises however is- can a software solution be implemented in a real multi-core environment? To answer this question we have to understand the environment in which the software solutiom can perform. For our final evaluation we have considered a processing system, thats a full fledged system on a chip. This is equipped with considerable on-chip generic resources (advanced caching mechanisms) which are quite cheap and inexpensive in modern chip solutions. More processing power continues to be integrated on to the modern chips with more expendable resources closer to each core. Application

specific co-processors are very frequently used to speedup applications and provide them with dedicated compute power. The question is in the future, can we offer a memory controller a dedicated processor and with some inexpensive resources? If the answer in modern as well as future solutions is yes, then the results discussed here show that we can in fact think of a feasible and highly customizable software solution.

REFERENCES

[1] 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.

[2] 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.

[3] Y. Rajasekhar and R. Sass, "A novel memory subsystem and computational model for parallel reconfigurable architectures," in *Euro-Par 2013: On-chip Memory Hierarchies and Interconnects (not yet available in print)*. Springer.

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

[5] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[6] 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.

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

[8] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*. ACM, 2004, p. 162.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[10] "Microblaze processor reference guide."

[11] "Zynq 7000 all programmable soc, technical reference manual."

[12] "Amazon fine foods reviews."

[13] S. Hawayek, "Feasibility study of using named memory segments instead of byte addressable memory in highly parallel many core systems," 2014.