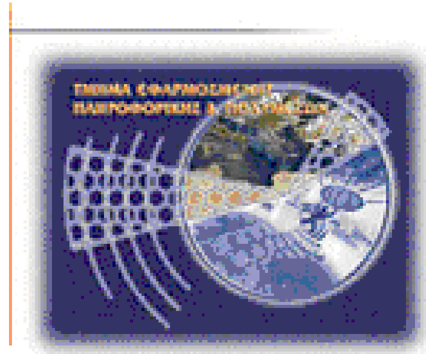
Technological Educational Institute of Crete

School of Engineering
Department of Informatics Engineering



GRADUATE THESIS

Design and Implementation Mechanisms for Quality of Service over Embedded System on Chips

Matthaiou Eustathia
A.M. 2930

Supervising Professor:
Kornaros George

July 2015– Herakleion

Acknowledgements

First of all I would like to express my gratitude to my supervising professor Mr. G. Kornaros for the trust and patience he showed me. I would also like to thank him for giving me the opportunity to work in this specific field in computer science and for his guidance and valuable help he provided me with throughout this process.

I would also like to thank my family and friends for supporting and bearing with me the whole time. Without their ethical support it would have been much harder for me to finish this thesis.

Moreover, I would like to dedicate this thesis to my father who has supported me and has always had my wellbeing in his mind. He has always told me that there is nothing I can't do and thanks to that I have managed to accomplish my goals.

Σύνοψη

Τα ενσωματωμένα συστήματα έχουν γνωρίσει μεγάλη ανάπτυξη τα τελευταία χρόνια. Η χρήση πολυπύρηνων επεξεργαστών, μέσα σε αυτά τα συστήματα, έχουν πραγματικά ενισχύσει την απόδοσή τους. Ωστόσο, η χρήση πολυπύρηνων επεξεργαστών δεν είναι πάντα απλή και μπορεί να δημιουργήσει δυσκολίες. Για να μπορέσουμε να παρατηρήσουμε και να βελτιστοποιήσουμε τα συστήματα, και πιο συγκεκριμένα τα Συστήματα σε Τσιπ (System on Chip), χρησιμοποιούμε Performance Monitors.

Ο κύριος στόχος της πτυχιακής μου εργασίας είναι να υλοποιήσουμε εφαρμογές για ποιότητα υπηρεσιών (Quality of Service) πάνω σε πολυπύρηνους επεξεργαστές σε Συστήματα σε Τσιπ. Σε κάποιες περιπτώσεις προσπαθούμε να πετύχουμε παραλληλισμό. Με την χρήση του Performance Monitoring Unit που παρέχεται , καταγράφουμε επιθυμητά συμβάντα ώστε να τα επεξεργαστούμε. Ο τελικός στόχος είναι να χρησιμοποιήσουμε αυτές τις πληροφορίες προς την διαχείριση πόρων.

Λέξεις Κλειδιά: Συστήματα σε Τσιπ, Μονάδα Παρακολούθησης Απόδοσης, ARM Cortex-A9, Ποιότητα Υπηρεσιών, Sobel, TEA, Black Scholes

Key Words: System on Chips, Performance Monitoring Unit, ARM Cortex-A9, Quality of Service, Sobel, TEA, Black Scholes.

Table of Contents

Acknowledgements 3
Σύνοψη 4
Table of Contents 6
List of Figures 9
List of Charts 10
List of Tables 11
Abstract 13
1. Introduction 14
1.1 Reasons for Conducting the Thesis 14
1.2 Related Work 15
1.2 Aims and Objectives of Thesis 15
1.3 Chapter Summary 15
2. The Performance Monitoring Unit 17
2.1 The ARM Performance Monitoring Unit 17
2.2 The Performance Monitoring Unit Counters 18
2.3 The Performance Monitoring Unit Events 18
2. 4The Performance Monitoring Unit Behavior on Overflow 19
2. 5The Performance Monitoring Accuracy 19
3. Applications/Benchmarks 20
3.1 Benchmarks in Embedded Computing 20
3.2 Sobel Application 21
3.3 TEA Application 21
3.4 Black Scholes Application 22
4. Hardware Implementation Description 23
4.1 The Zynq-7000 AP SoC 23

4.2 The Processing System 24
4.2.1 The ARM Architecture 24
4.2.2 The ARM Coretex-A9 Processors 25
4.2.3 The Cache Memory 26
4.2.3.1 The CPU Cache 27
4.2.3.2 The Level-One (L1) Cache 27
4.2.3.3 The Instruction Cache (I-Cache) 28
4.2.3.4 The Data Cache (D-Cache) 28
4.2.3.4 The Level-Two (L2) Cache 29
4.2.3.5 The Exclusive L2 Cache 29
4.2.4 Cache Coherency 29
4.2.4.1 The Snoop Control Unit (SCU) 30
4.2.5 The Memory Unit 30
4.2.5.1 The Memory Types 31
4.2.5.2 The DDR Memory 31
4.2.5.3 The DDR Controller 31
4.2.6 The ARM Timers 31
4.2.6.1 The Private Timer 32
4.3 The Programmable Logic 32
4.3.1 The Advanced Microcontroller Bus Architecture (AMBA) 32
4.3.1.1 The Advanced Extensible Interface (AXI) 33
4.3.1.2 The AXI BRAM Controller 33
4.3.1.3 The AXI GPIO 33
4.3.2 The Random Access Memory (RAM) 33
5. Software Implementation Description. 35
5.1 Single Sobel and TEA Application 35
5.2 Multiple Sobel and TEA Application 36
5.2 Performance Monitoring Unit Application 37
6. Measurements-Results 39
6.1Formulas 39
6.2Sobel and TEA Applications 40
6.2.1Sobel and TEA Application Time 40
6.2.2 Sobel and TEA Instruction Cache Miss Bandwidths 40
6.2.3 Sobel and TEA Data Cache Miss Bandwidths 41
6.2.4 Sobel and TEA Data Cache Access Bandwidths 43

6.2.5 Sobel and TEA Load/Store Instructions Bandwidths ... 44
6.2.6 Sobel and TEA Cache Ratio ... 45
6.3Black Scholes Application ... 46
7. Conclusions and Future Work ... 50
7.1 Future Work ... 50
Bibliography ... 52
Appendix A ... 53
1.1 Additional Performance Monitoring Events ... 53
1.1.1 Implemented architectural events ... 53
1.1.2 Coretex-A9 Specific Events ... 54
1.2 PMU Assembly Access Functions ... 55
Appendix B ... 59
1.1 Additional Measurements ... 59
1.1.1 Single Sobel and TEA Application ... 59
1.1.2 Multiple Sobel and TEA Application ... 61
Appendix C ... 63
1.1 Thesis Presentation ... 63

List of Figures

Figure 3-0-1: Sobel 3x3 Masks 21
Figure 3-0-2: Two Feistel rounds (one cycle) of TEA 21
Figure 4-0-1: The Zynq-7000 AP SoC Overview 24
Figure 4-0-2 The APU Block Diagram 26
Figure 5-0-1: While Loop In TEA Application 35
Figure 5-0-2: Image After Sobel Filter 36
Figure 5-0-3: Original Image 36
Figure 5-0-4: Process of Multiple Sobel and TEA Applications 36
Figure 5-0-5: Enabling Functions 37
Figure 5-0-6: Counting Functions 37
Figure 5-0-7: Disabling Functions 38

List of Charts

Chart 1 : Sobel and TEA Application Times 40
Chart 2: The Instruction Cache Miss Bandwidths for the Sobel Application 41
Chart 3: The Instruction Cache Miss Bandwidths for the TEA Application 41
Chart 4: The Data Cache Miss Bandwidths for the Sobel Application 42
Chart 5: The Data Cache Miss Bandwidths for the TEA Application 42
Chart 6: The Data Cache Access Bandwidths for the Sobel Application 43
Chart 7: The Data Cache Access Bandwidths for the TEA Application... 43
Chart 8: The Load/Store Instructions Bandwidths for the Sobel Application 44
Chart 9: The Load/Store Instructions Bandwidth for the TEA Application 45
Chart 10: The Cache Ratio for the Sobel Application 45
Chart 11: The Cache Ratio for the TEA Application.. 46
Chart 12: The Data Cache Miss Bandwidth for all five of the Black Scholes Applications 47
Chart 13: The Data Cache Access Bandwidth for all five of the Black Scholes Applications............. 47
Chart 14: The Load/Store Instructions Bandwidth for all five of the Black Scholes Applications...... 48
Chart 15: The Cache Ratio for all five of the Black Scholes Applications 48
Chart 16: The Times For the MC Test.. 49

List of Tables

Table 1: Single Sobel With L1 Cache Disabled 59
Table 2: Single TEA With L1 Cache Disabled 59
Table 3: Single Sobel with L2 Cache Disabled 59
Table 4: Single TEA With L2 Cache Disabled 60
Table 5: Single Sobel With I cache Disabled 60
Table 6: Single TEA With I Cache Disabled 60
Table 7: Single Sobel With D Cache Disabled 60
Table 8: Single TEA With D Cache Disabled 61
Table 9: Multiple Sobel and TEA With L1 Cache Disabled 61
Table 10: Multiple Sobel and TEA With D Cache Disabled 61
Table 11: Multiple Sobel and TEA With I Cache Disabled 62
Table 12: Multiple Sobel and TEA With L2 Cache Disabled 62

Abstract

Embedded systems have seen great growth in recent years. The usage of multi-core processors, within these systems, has truly enhanced their performance. However, using multi-core processors is not always simple and it can create challenges. In order to observe and optimize the systems, and to be more exact the System on Chips (SoCs), we employ the Performance Monitors.

The main focus of this thesis is to implement applications for quality of service (QoS) on multi-core SoCs. In some cases these applications try to reach parallelization. With the usage of the Performance Monitoring Unit provided, we capture desired events so as to examine them. The ultimate goal is to use this information towards resource management.

1. Introduction

In this thesis we designed a system, on which various applications are implemented so as to monitor the system and its resources. The design involves a system with multi-core processors (two cores). Most of the applications are implemented in a way, to achieve simultaneous processor occupation. We then enable the hardware Performance Monitors so as to study events performed on the system and its resources. The software applications were written with the usage of the C programming language.

Our architecture was built on the Zynq-7000 All Programmable SoC and designed with the Xilinx Vivado Design Suite. The applications were implemented and programmed with the Xilinx Software Development Kit (SDK).

The Performance Monitoring Unit (PMU) is contained in the processors debug architecture. We use the system control coprocessor interface (CP15) in order to access it. The PMU is enabled and captures certain events that we define, before the application starts and after the application has ended. The results calculated are then compared in pursuance of drawing conclusion referring to the system and its resources.

1.1 Reasons for Conducting the Thesis

Multi-core designs have become widespread and for this reason they are the object of extensive discussion and research. More importantly, research is done on the subject of energy consumption, resource utilization and overall system performance.

The main use of the Performance Monitoring Unit is to accurately monitor the performance of the system under certain circumstances and to provide the program developer with this information. This information constitutes a sort of guideline helping the developer improve the system with the appropriate changes.

Even though there is research on these multi-core embedded systems, not many researchers have explored the use of the Performance Monitoring Unit provided by the ARM Debugging architecture through the system control coprocessor interface (CP15).

By using the Zynq-7000 and exploring the resource utilization through the Cortex-A9 processors we hope to better interpret how the processing system functions, with the ultimate aim to exploit the system to our full advantage.

1.2 Related Work

Upon energy consumption in [1] CASHIER, a Cache Energy Saving Technique for Quality of Service Systems, is presented. Here Cashier uses dynamic profiling to estimate the memory subsystem energy and execution time of any program under multiple last level cache (LLC) configurations. It then reconfigures LLC to an energy efficient configuration with a view to meet the deadline. It manages to balance the energy saving and performance loss by adapting itself thus supporting higher performance for a larger cache. Power consumption is addressed in [2], with the usage two different algorithms. The first algorithm dynamically balances the task load for multiple cores so as to optimize energy during execution. The second algorithm, which is the Dynamic Core Scaling Algorithm, adjusts the number of active cores to reduce power leakage. Both these algorithms have proven to conserve up to 25 percent and 40 percent of the energy respectively.

A similar approach on Performance monitoring but with a different system (in this case NoCs instead of SoCs) is suggested in [3]. Hardware agents are deployed to monitor managers that can be dynamically configured and that can calculate statistics. This methodology that programs hardware units, addresses the real time monitoring to improve the quality of service and the resource management. The incorporation of the monitors in these designs confirms that they are of small prerequisite and are beneficial in dynamic resource management.

A more close appeal to what we study is recommended in [4]. An on-chip bus PMU is utilized, which accurately evaluates the system power consumption. This design is customized for different on-chip and off-chip memory devices and does not dependent on a specific CPU core. The memory devices that use energy state machines are described in XML. This PMU traces the internal behavior of the memory devices to give a estimate of power consumption.

1.2 Aims and Objectives of Thesis

In a summary, the aims of this thesis are the following:

- Implementation of applications on the hardware design
- Enablement of the Performance Monitoring Unit
- Measurements by the Performance Monitoring Unit on the effect the applications have on the system.
- Study in the results obtained

1.3 Chapter Summary

Chapter 1: An introduction to the thesis

Chapter 2: An extensive description on the Performance Monitoring Unit

Chapter 3: A presentation on the Applications/Benchmarks used

Chapter 4: The description of the Hardware design in use

Chapter 5: A display of the software implementation

Chapter 6: A presentment of the results obtained

Chapter 7: Conclusions drawn from the measurements and ideas towards future development

2. The Performance Monitoring Unit

The Performance Monitor Unit, or else known as the PMU, is found in all high end processors these days. [5] It provides the application developers the ability to measure predefined events and processor clocks related to specific operations, for instance cache misses and CPU stalls, in the interest of counting the efficiency of their application software.

The PMU is essentially hardware that has been built inside any give processor in order to measure its performance parameters. It has a tight integration with the CPU core, meaning that every CPU has its own Performance Monitor Unit. The PMU was originally designed by computer hardware engineers for the use of debugging CPUs that is why it is also known as Hardware Performance Counters (HPCs).

Since the Performance Monitor Unit has a hardware implementation, we can expect very limited overhead. It does not use any of the computational or storage resources that are needed for normal operations of the CPU thus providing low perturbation. The high resolution presented by the PMU enables the monitoring of detailed micro-architectural events that in any other case would not be monitored without hardware support. The most important advantage of the existence of Performance Monitor Units is that they are widespread. Mostly all of the dominant industrial processors have included them in their designs.

2.1 The ARM Performance Monitoring Unit

When writing optimized code, having knowledge of the processors behavior with branches can be more than useful. For that reason branch prediction is considered to be part of the hardware implementation. The performance monitor counters generate information regarding the number of branches that are correctly or incorrectly predicted and are used to profile and in most cases benchmark code [6].

The Performance Monitor Unit (PMU) is part of the ARM Debug architecture. Before the existence of the ARMv7 processors, the performance monitors were included but not part of its architecture. According to the ARMv7 [7], the system control coprocessor interface (CP15) is a mandatory interface for the Performance Monitor registers. Other possible interfaces for the Performance Monitor registers are a memory-mapped and an external debug interface, which both are optional.

By using the CP15 interface, an operating system running on a processor can enable access to counters within the application software. Therefore the application is able to monitor itself. In many cases ARM recommends implementing application software access to the Performance Monitors when the operating system does not use the monitors. The CP15 supports the usage of

energy management and dynamic compilation techniques. In conclusion ARMv7 reserves the CP15 registers strictly for ARM-recommended and implementation defined Performance Monitors.

The Performance Monitors main form consists of a cycle counter that is able to count either every cycle or configured to count every 64th cycle, a number of 32-bit wide event counters which are programmable and controls that are used to enable and reset counter, flag overflows and enable interrupts on an overflow.

2.2 The Performance Monitoring Unit Counters

The counters, a performance counter block contains, may be accessed through debug tools or through software that runs on the processor, by using the CP15 Performance Monitoring Unit registers. This feature is non-invasive and does not change the behavior of the processor. The Cortex-a9 Performance Monitoring Unit provides six event counters (PMU event counter 0 to 5) to calculate statistics on the operations the processor performs and on the memory system [8]. Each on of the counters are able to count any of the 63 events available in the Cortex-a9 processor. The results given by the counters are approximate and all of the counters are subject to any changes in clock frequency. The monitoring software can enable the cycle counter independently in comparison to the other event counters, and its only control over this counter is an access permission control for User mode.

2.3 The Performance Monitoring Unit Events

The events that one may monitor can be divided into categories, the architectural/micro-architectural events and the implementation-specific events. The events are identified with the usage of an event number assigned to each event.

The events that are used in this thesis are both architectural and specific. The architectural events we use are: Instruction cache miss (0x01), Data cache miss (0x03), Data cache access (0x04), Data read (0x06), Data writes (0x07) and Cycle count (0x11). The specific events we use are: Load/Store instructions (0x72) that counts the number of instructions being executed in the Load/Store unit, Processor stalled because of a write to memory (0x81) that counts the number of cycles when the processor is stalled, Processor stalled because of instruction side main TLB miss (0x82) that counts the number of stall cycles because of main TLB misses on requests issued by the instruction side and Processor stalled because of data side main TLB miss (0x83) that counts the number of stall cycles because of main TLB misses on request issued by the data side. The information generated by these specific events is approximate.

2. 4The Performance Monitoring Unit Behavior on Overflow

The Performance Monitoring Unit counts events with the usage of 32-bit wrapping counters. When a counter overflows it will wrap. Upon this case an overflow status bit is set to the value 1. If the processor is configured to generate counter overflows, an interrupt request will be generated. Last but not least, on a Performance Monitor counter overflow the counter will proceed to counting events.

2. 5The Performance Monitoring Accuracy

The information generated by the Performance Monitors is approximately accurate. This reasonable degree of inaccuracy provided by the counters is acceptable although it is not defined by ARM. However ARM does recommend following guidelines such as: under normal operating conditions the counters must present accurate counts, in extraordinary situations an inaccuracy in counter value is respectable and in asynchronous exceptions, for instance interrupts, the counts may be inaccurate.

The permitted inaccuracy does in some way limit the possible use of the Performance Monitoring Unit. Cases which contribute to the imprecise results of the Performance Monitor are pipelining, change in the security state and entry to and exit from the Debug state. An implementation that can somewhat limit counter imprecision to a certain extent is disabling the counters as soon as possible during the Debug state entry sequence. By any means, an implementation should document scenarios where inaccuracies are expected.

3. Applications/Benchmarks

This chapter is a reference to the types of Applications/Benchmarks that will be implemented and used, in this thesis, along with the Performance Monitoring Unit to monitor our system, its performance and its resource management.

3.1 Benchmarks in Embedded Computing

A Benchmark, in terms of computing, is the performance of running numerous standard tests and trials (computer programs, a set of programs and operations) against an object, so as to assess its relative performance [9]. They provide a method of comparing the performance on assorted subsystems across different chip or system architectures. "Benchmark" as a term is also used to describe elaborately designed benchmarking programs themselves.

Benchmarking is mostly correlated with estimating the performance characteristics of computer hardware, but there are also situations in which the technique is applied to software. In these types of software Benchmarks may run against compilers or even database management systems.

The main purpose for the development of these tests, were to compare different architectures, since with the passing of time the computer architectures have advanced and the comparison between the performances of various computer systems has become more difficult.

Benchmarks are designed to simulate a type of workload on a component or a system. A synthetic workload is simulated by uniquely creating programs that impose the workload on the component. On the other hand, Benchmarks applications are used to run real-world performance on a system.

Benchmarking is not always simple and it often involves different iterative rounds so as to reach predictable and useful conclusions [10]. Interpreting benchmarked data is also extremely difficult. Many benchmarks focus solely on the speed of computational performance while overlooking other important features such a quality of service. An example of unmeasured quality of service features includes security, availability, reliability, execution integrity, serviceability and scalability.

Various types of Benchmarks exist. Such are: real program, component Benchmark or micro benchmark, kernel, synthetic Benchmark (which include the well known Whetstone and Dhrystone), I/O benchmarks, database benchmarks and parallel benchmarks.

3.2 Sobel Application

The Sobel filter, also known as the Sobel operator, is one of the most known methods that is used in image processing and for computer vision [11]. In particular it is an algorithm that detects edges and subsequently creates an image where the edges and transitions are emphasized.

The result given by this filter is a new binary image, where the pixels with the greatest value are now the edges of the original image. Nonetheless, the dimensions of the image remain the same. After this process, thresholding follows which usually consists of maintaining a percentage of the edges that have a higher gradient.

The Sobel technique emphasizes on areas with high spatial frequency. It is applied with the use of an operator that consists of two "edge masks", so as to detect changes vertically and horizontally. These two masks are two 3x3 convolution kernels. The convolution between the two masks and the image is carried out throughout the edge detection. By combining the two pictures that occur, the edges of the images object emerge [12].

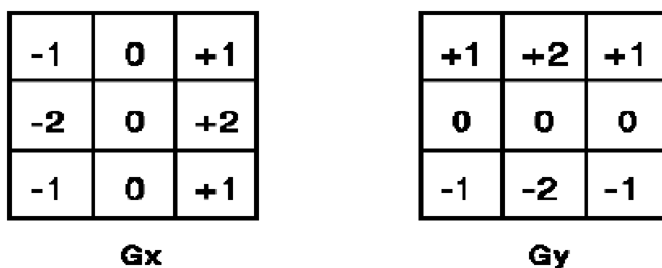


Figure 3-0-1: Sobel 3x3 Masks

3.3 TEA Application

The Tiny Encryption Algorithm is one of the fastest and most efficient block ciphers in existence. It was designed by David Wheeler and Roger Needham of the Cambridge Computer Laboratory. It operates on two 32-bit unsigned integers and uses a 128-bit key [13].

The Feistel structure, in which the TEA algorithm is implemented, consists of 64 identical rounds that contain function bits for translocations, mod 2^8 additions or subtractions and the exclusive-or (XOR) calculation. It has a simple key schedule by mixing all of the key material in the exact same way for each cycle.

Although TEA seems to be extremely resistant to differential cryptanalysis, and achieves comprehensive diffusion after just six rounds, it has a few "weak spots". First and foremost, it suffers from equivalent keys. To be exact, each key is equivalent to three other keys, making the effective key size to be only 126 bits. This results in TEA

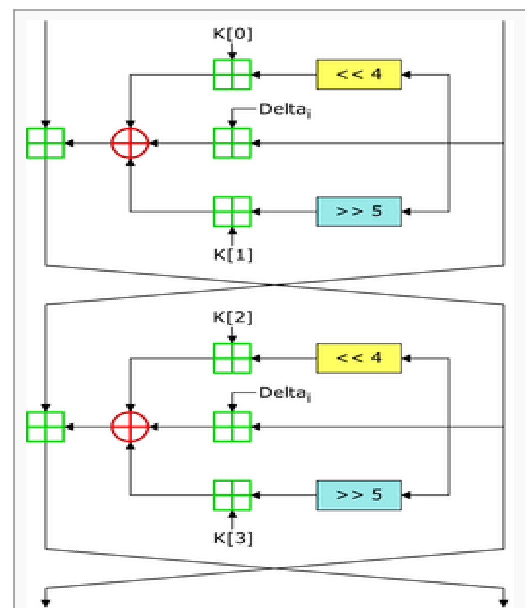


Figure 3-0-2: Two Feistel rounds (one cycle) of TEA

being unfit as a cryptographic hash function. It is also receptive to a related-key attack. In this case, 223 chosen plaintexts under a related-key pair are required, with a 232 time complexity.

3.4 Black Scholes Application

The Black-Scholes –Merton model, widely known as the Black-Scholes model, is a mathematical model of a financial market that contains derivative investment instruments. It is a model of price variation over time of financial instruments, such as stocks, that can be used to determine the price of a European call option. The Black-Scholes model is one of the most important concepts in the field of modern financial theory. It was developed by Fisher Black, Robert Merton and Myron Scholes (hence its name) in 1973. Up to this day it is widely used and viewed as one of the best way to determine the fair price of options.

The Black-Scholes workload computes the Black-Scholes formula for European and call options in terms of five parameters: the spot price of the underlying stock, the exercise price at which the transaction will be executed, the expiration period after which the option can be exercised, the risk-free rate of return and the volatility of returns of the underlying stock.

Black-Scholes is part of the PARSEC Benchmark suit [14]. PARSEC contains thirteen applications, each of which are used for a specific area of interest. Each application workload is parallelized in multiple ways in order to enable various benchmark studies. The Black-Scholes model is data-parallel and includes a list of routines such as: ASSET_PATH that simulates the behavior of an asset price over time, BINOMIAL that uses the binomial method for a European call, BSF that evaluates the Black-Scholes formula for a European call, FORWARD that use the forward difference method to value a European call option and MC that uses Monte Carlo valuation on a European call

4. Hardware Implementation Description

4.1 The Zynq-7000 AP SoC

In terms of hardware, for the implementation of this present thesis, our architecture was built based on the Zynq-7000 architecture [15]. The Zynq-7000 integrates a feature-rich dual-core ARM Cortex-A9 MPCore based processing system and Xilinx programmable logic in a single device. The ARM CPU is the heart of the processing system which also includes on-chip memory, external memory interfaces, and a rich set of I/O peripherals. The various hardware controllers as well as the processing system are I/O interconnected via high-bandwidth AMBA AXI interfaces.

In the Processing System, the processors are the ones to boot first, thus allowing a software centric approach for the Programmable Logic system boot and the Programmable Logic configuration. The Programmable Logic can be configured as part of the booting process or it can be configured later at some point. It can also, in addition, be altogether reconfigured or used with partial and dynamic reconfiguration. The data used to configure the Programmable Logic is mostly referred to as a Bitstream.

The Zynq-7000 AP SoC is composed of two major functional blocks: the Processing System (PS) and the Programmable Logic (PL). The Processing system consists of an Application Processor Unit (APU) that provides an extensive offering of high-performance features and standards-compliant capabilities, memory interfaces, I/O peripherals (IOP) and interconnects. The Programmable Logic is derived from the Xilinx 7 series FPGA technology.

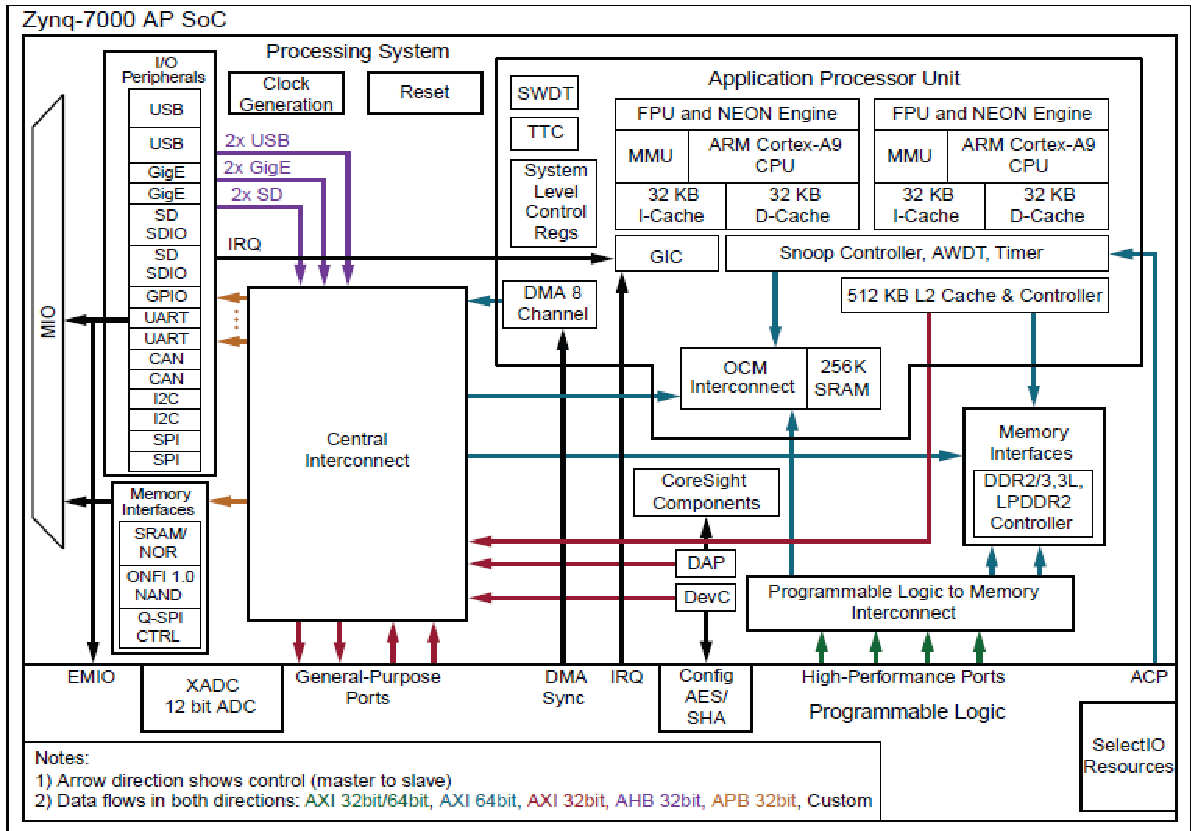


Figure 4-0-1: The Zynq-7000 AP SoC Overview

4.2 The Processing System

4.2.1 The ARM Architecture

The ARM architecture endorses the implementation through a variety of performance points. The simplicity this architecture has to offer has led to small implementations, thus allowing these implementations to be used by devices with very low power consumption. Key attributes in the development of the ARM Architecture are the implementation size, the performance and the low power consumption.

The ARM architecture is a Reduced Instruction Set Computer (RISC) and it assimilates certain RISC architectural features such as : a large uniform register file, a load/store architecture where data-processing operations only operate on register contents not directly on memory contents, simple addressing modes with all the load/store addresses being determined from register contents and instruction fields only and uniformed and fixed-length instruction fields so as to simplify instruction decoding.

Additionally the ARM provides features such as: control over both the Arithmetic Logic Unit (ALU) and the shifter that is associated with instructions, auto-increment and auto-decrement addressing modes to optimize program loops, loading and storing multiple instructions in order to maximize the execution throughput and restrictive execution of almost all the instructions to

maximize the execution throughput. These characteristics incorporated in the RISC architecture contributes in the ARM processors having a good balance of high performance, small code size, low power consumption and small silicon area.

4.2.2 The ARM Coretex-A9 Processors

The dual-core Cortex-A9 configuration is implemented by the APU. Each one of the processors have their own SIMD media processing engines (NEON),memory management units (MMU), 32 KB level-one (L1) data caches and 32 KB level-one (L1) instruction caches, private timers and watchdog timers. These two Cortex-A9 processors provide two 64-bit AXI master interfaces for independent data and instruction transactions, which can then be routed to the on-chip memory (OCM), the 512kb sharable level-two (L2) cache, the DDR memory or through the processing systems interconnect to other slaves in the processing system (PS) or to the programmable logic (PL). The processors run time options acquiesce single processor configurations and asymmetrical (AMP) or symmetrical (SMP) multiprocessing configurations.

The Cortex-A9 processor enables fundamental hardware features for program debugging. With that been said, it also provides hardware counters to assemble information on specific operations of the processor and the memory system. Each processor is able to issue two instructions in only one cycle whilst executing them out of order. Other characteristics of the Cortex-A9 features amongst others is: a superscalar variable length pipeline with dynamic branch prediction, a full implementation of the ARM architecture v7-A instruction set, an execution of 32-bit ARM instructions, an execution of 16-bit and 32-bit Thumb instructions, an execution of 8-bit Java byte codes in Jazelle hardware state, security extensions and support for advanced power management with up to three power domains.

The ARM architecture provides 31 general-purpose 32-bit registers of which the 16 are visible at any given time. The registers that cannot be accessed are used to speed up the processing. These processors support byte (8 bits), word (32 bits), halfword (16 bits), doubleword (64 bits) which are data types in memory. Load and store instructions can transmit bytes, halfwords or words to and from the memory. The instruction set includes load and store operations the can transfer more than two words to and from the memory. By using these instructions, the software can load and store doublewords.

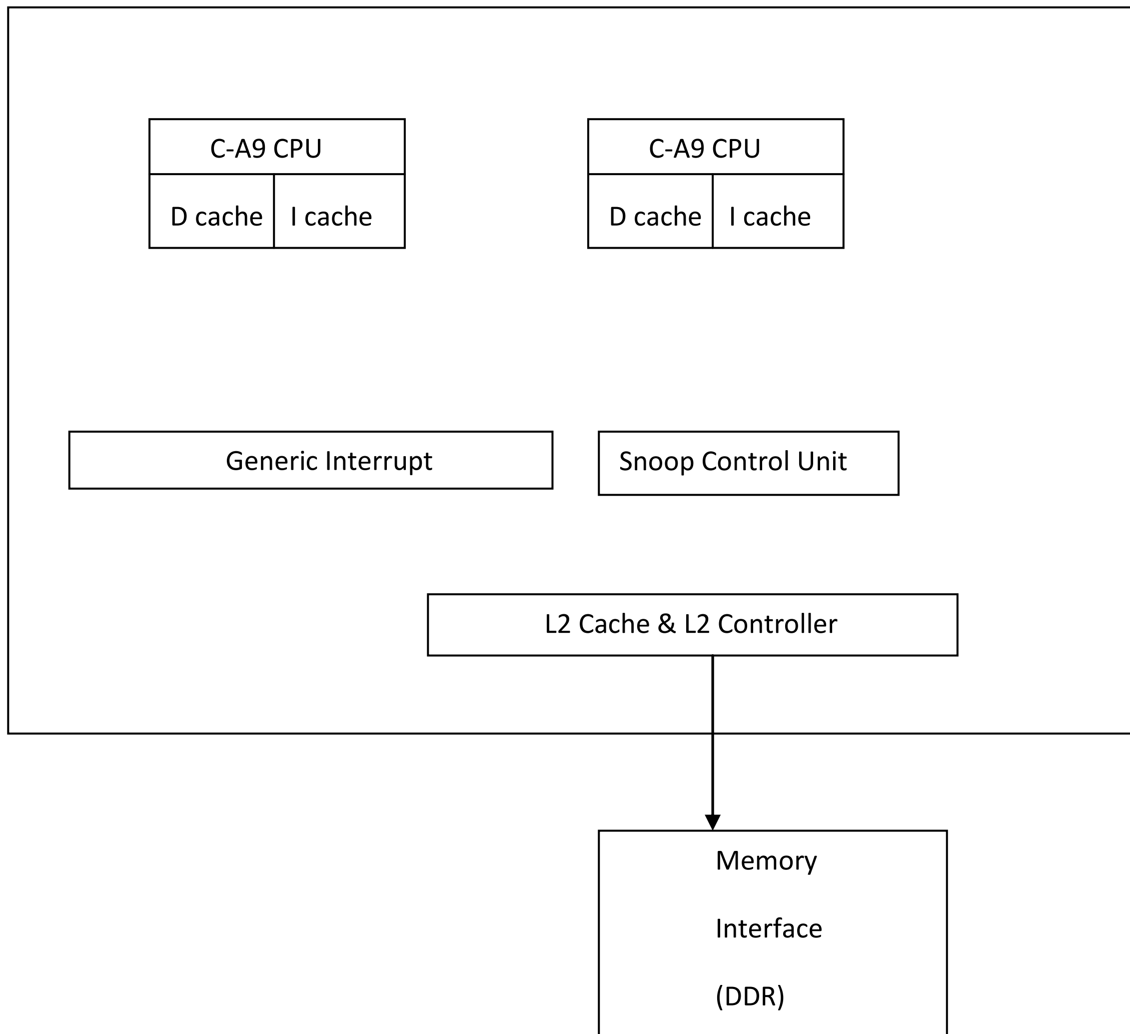


Figure 4-0-2 The APU Block Diagram

4.2.3 The Cache Memory

In terms of computing, a Cache is a type of memory that stores data for future use where, when requested, the data is accessed faster. This data stored in the cache memory might be the equivalent of data stored somewhere else in the system or the results of a computation made earlier. Caches are comparatively small, thus achieving cost-effectiveness and efficient use of data. Nonetheless, due to the access patterns in typical computer applications, which exhibit the locality of reference, caches have proven to be very important and useful in many areas of computing. There are two types of access patterns exhibited, temporal locality and spacial locality. The temporal locality pattern refers to requested data that have already recently been requested, whereas the spacial locality pattern is associated with requests for data that have been physically stored near to data that have already been requested.

A cache consists of a number of entries. Each entry has a piece of data, which is a copy of the data that is stored in some other storage unit. Apart from the data, it also has an entry tag that determines the identity of the data. When the cache client wants to access data that presumably exists in some memory unit, it first goes through the cache to check. If an entry matching the tag of the requested data is found, then that data is used instead. This event is referred to as Cache Hit. The event that occurs when that data cannot be found in the cache is referred to as Cache Miss. In this case the "uncached" data is acquired from the memory unit and copied to the cache for future access.

During a cache miss, the CPU usually ejects some other entry in order to make room for the previously "uncached" datum. The method used to select the data that will be ejected is known as the Replacement Policy. The Least Recently Used (LRU) is a popular replacement policy that replaces the least recently used entry.

A cache memory is much faster than a main memory for various reasons. The use of rapid electric circuits, which leads to higher expenditures regarding costs, size and power requirements. Due to the fact that the memory is small, the increase in cost is relatively limited. A cache memory has fewer areas in comparison to a main memory, thus resulting in a shallower decoding tree which contributes in reduced access size. The cache memory is static in contrast to the main memory which is mostly dynamic. It is placed naturally and logically closer to the CPU as opposed to the main memory that results in the prevention of delays due to the shared communication channel (shared bus).

In conclusion, reading data from a cache memory is faster than recomputing a result or reading data from a slower data store, thus, the more requests served from the cache, the faster the performance of the system. Central processing units (CPU) and Hard Disk Drives frequently use a cache.

4.2.3.1 The CPU Cache

A cache used by the Central Processing Unit (CPU) of a computer is a CPU Cache used to reduce the average time of accessing data from the main memory. When the processor wants to read from or write to a location in the main memory, it checks first whether a copy of that data exists in the cache. If the data does exist, the processor reads from or writes to the cache, making it much faster than reading from or writing to the main memory.

Most CPUs have at least three different caches which are the Instruction Cache, the Data Cache and the Translation Lookaside Buffer even though it is a part of the MMU and not directly related to the CPU caches. The Instruction cache is used to speed up executable instruction fetches. The Data cache is used to speed up data fetches and stores and the TLB is used to speed up virtual-to-physical address translations for both executable instructions and data. The Data caches are organized as a hierarchy of more caches, the level one (L1) and the level two (L2).

4.2.3.2 The Level-One (L1) Cache

The level one (L1) Cache is the nearest level of cache to the CPU. It can be implemented in a Harvard arrangement or in a von Neumann arrangement. In the first case the Instruction Cache and the Data Cache are separate whereas in the second case all the cache items are unified. An implementation with a Harvard arrangement does not necessarily have to include hardware support for coherency amongst the Data and Instruction caches.

The two Cortex-A9 processors both have separate 32 KB Level-1 Instruction and Data caches. Each cache can independently be disabled by using the system control coprocessor. Both L1 caches are 4-way set-associative with 32 byte cache line lengths and support parity. These caches support 4 KB, 64 KB, 1 MB, and 16 MB virtual memory page however neither of the two L1 caches

support the lock-down feature. In the case of a cache miss, the cache replacement policy is either pseudo round-robin or pseudo-random and a critical word first filling of the cache is performed. During implementation, the Level 1 Instruction and Data cache can independently be configured to the sizes of 16KB, 32KB, or 64KB. In order to reduce the power consumption, the number of full cache reads is reduced by exploiting the sequential nature of many cache operations. When a CPU reset occurs, the contents of both L1 caches are cleared to comply with security requirements.

All the memory attributes are exported to the external memory systems.

4.2.3.3 The Instruction Cache (I-Cache)

The Level 1 Instruction Cache (I-Cache) is responsible for administering an instruction stream towards the processors. The Instruction cache interfaces precisely to the pre-fetch unit that consists of a two-level prediction mechanism. It is virtually indexed and physically tagged. The replacement policy for the I-cache is either pseudo round robin or pseudo random.

4.2.3.4 The Data Cache (D-Cache)

The Level 1 Data cache (D-Cache) is responsible for containing the data which the processor uses. The Data cache is physically indexed and physically tagged, as is the Instruction cache. It is non-block, meaning that the load/store instructions may proceed in hitting the cache while it is performing allocations from the external memory because of prior read or write misses. This type of cache supports four outstanding reads and four outstanding writes. The Data cache also supports two 32-byte line-fill buffers and one 32-byte eviction buffer, while the Cortex-A9 CPU has a store buffer featuring four 64-bit slots with data merging capability. The write-back and write-allocate policy are only supported by the Data cache. The write-through and write-back/no write-allocate policies are not implemented. The Level 1 Data cache supports exclusive operations, implying that the cache line is valid only in Level 1 or Level 2 cache and never in both simultaneously, with respect to the Level 2 cache. If the exclusive operation is disabled by default, the cache utilization will then increase and the power consumption will be reduced. The replacement policy for the D-cache is pseudo random.

4.2.3.4 The Level-Two (L2) Cache

The Level 2 Cache (L2-cache) is an 8-way set associative cache with a size of 512 Kb used for dual Cortex-A9 processor cores. The L2 cache can either be tightly coupled to the core or implemented as memory mapped peripheral on the system bus. In the memory-mapped case, where cache control functions require an address parameter, for example, clean entry by address, the address must be inherently a physical address. Level 2 caches that are more closely coupled to the core can use virtual or physical addresses. It is physically tagged and physically addressed and supports a fixed 32-byte line size. A parity check is offered for the Level 2 cache. To improve the latency, a critical –word-first-line-fill is supported. In the case of cache miss, the selection policy implemented is pseudo random with deterministic option. This cache supports write-though, write-back, read allocate, write-allocate and read and write allocate.

The L2 controller implements multiple 256-bit line buffers, two for each slave port, to improve cache efficiency. These buffers hold a line from the L2 cache in case of a cache hit. The L2 cache controller also implements three 256-bit eviction buffers that hold the evicted lines from the L2 cache, in order for them to be written back to the main memory as well as three 256-bit store buffers to hold “bufferable” writes before their drainage to the main memory, or to the L2 cache. These buffers enable multiple writes to the same line so as to be merged. Another characteristic featured by the L2 cache controller is that it is able to forward exclusive requests from L1 to DDR, OCM, or the external memory.

4.2.3.5 The Exclusive L2 Cache

The Level 1 and Level 2 cache provide an exclusive mode. This mode has to be activated both in the processor and the current cache controller that is being occupied. When the L2 cache is used, the data cache of the processor and the L2 cache are exclusive. This means that at any time, a given address is cached in either the L2 data cache or the L1 data cache, but not in both. This mode increases the usable space and the efficiency of the L2 cache that is connected to the processor. When the exclusive cache mode is activated the data cache line replacement policy is modified so that the victim line is always evicted to L2 memory, even if it is clean.

4.2.4 Cache Coherency

The system preserves multiple versions of a value of a memory location when a cache or a write buffer is used. Assuming that Harvard caches are being used, both the Instruction and Data cache may contain a value of the memory location. Not all of these physical locations necessarily contain the value most recently written to the memory location. The real coherency issue is to secure that when a memory location is read, either by the Instruction cache or the Data cache, the value that is obtained, is at all times the value that was written to the location most recently. This can be difficult when there are multiple possible physical locations, such as main memory and at least one of a write buffer and one or more levels of cache. Some prospects of the memory system coherency, in the ARM architecture, are provided automatically by the system whereas other prospects are dealt with memory coherency rules. If a program breaks a memory coherency rule, the behavior that this program might cause is uncertain.

Address mapping and caches demand such a management in order to always ensure memory coherency. A cache and write buffer management require a sequence of: cleaning the data cache if it is a write-back cache, invalidating the data cache and the instruction cache, draining the write buffer, performing a pre-fetch flush on the instruction pipeline and flushing the branch prediction logic.

4.2.4.1 The Snoop Control Unit (SCU)

The two Cortex-A9 processors are organized with an MP configuration that contains a Snoop Control Unit (SCU). The SCU block connects the two processors to the memory system and incorporates data which manages the data coherency between the two processors and the Level 1 and Level 2 caches. This block is responsible for managing the interconnect arbitration, communication, cache and system memory transfers, and cache coherence for the Cortex-A9 processors.

The SCU block communicates with each of the Cortex-A9 processors through a cache coherency bus and manages the coherency between the L1 and the L2 caches. The SCU supports MESI snooping which provides increased power efficiency and performance by avoiding unnecessary system accesses. It also is enabled to check if there is data in the Level 1 cache by using great speed so as not to interrupt the processors. The SCU is also able to copy clean data from one processors cache to another processors cache therefore eliminating the need to access the main memory to perform this task. Moreover it can move dirty data between the processors, so as to skip the shared state and avoid the latency that is associated with the write-back.

4.2.5 The Memory Unit

The memory unit is the most vital component of any given computer. The ideal structure of a computer, in order to serve its needs, would consist of just one memory that would be rapid and vast. But in fact, as the demands on memories increases, such a thing seems impossible.

Due to this, the memory is organized in levels, thus the memory hierarchy. The numbers of levels that constitute a memory are derived from the systems needs. Generally, the higher we climb the chain the higher the performance increases and so does the cost. This results in implementing memories of smaller capacity in order to maintain a balance amongst the cost and the performance. The memory closest to the processor has a very low latency, but is limited in size and expensive to implement. On the other hand, the further from the processor, the easier it is to implement lager blocks of memory. However, these blocks have an increased latency.

The memory hierarchy of a computer system, starting low-level, consists of: processor registers and cache memories which are static memories, dynamic memories which constitute the main memory of a system, storage devices and optical storage disks.

By combining these types of memories from different levels and by using mechanisms, where the data that is most recently used is stored in higher levels in the hierarchy, an impression of a faster memory is given.

4.2.5.1 The Memory Types

For each memory region, the most significant memory attribute specifies the memory type. Therefore, three exclusive memory types exist: the normal memory type, the device memory type and the strongly-ordered memory type. Normal and device memory types have additional attributes. With the usage of the normal memory attribute, memory used for programs and data storage can be accessed. Memories that take advantage of the normal memory attributes are the programmed Flash ROM, the ROM, the SRAM, the DRAM and the DDR memory.

4.2.5.2 The DDR Memory

The Double Data Rate-Synchronous DRAM is a type of SDRAM. It is a class of memory integrated circuits which are used in computer systems. The DDR SDRAM interface makes higher transfer rates possible by more strict control of the timing of the electrical data and clock signals. The interface uses double pumping which transfers data on both the rising and falling edges of the clock signal in order to lower the clock frequency. By keeping the clock frequency at a low rate the signal integrity requirements on the circuit board are reduced and the memory is connected to the controller. Due to the double pumping the DDR SDRAM, with a certain clock frequency, it achieves nearly twice the bandwidth of a SDR SDRAM running at the same clock frequency.

4.2.5.3 The DDR Controller

The DDR Controller supports the DDR3, DDR3L, DDR2, and LPDDR-2. In our thesis, we use the DDR3 memory. The DRAM bus width averages from 16 bits to 32 bits and the burst length is 8. The rate of the controller is determined by the speed and the temperature grade of the device. It uses Data read strobe auto-calibration and enables a write data byte which is supported for each data beat.

4.2.6 The ARM Timers

In the ARM design, each one of the Cortex-A9 processors has their own Private 32-bit timer and Watchdog timer which are also 32-bit. A Global 64-bit timer also exists and is shared by both processors. Overall these times are always clocked at 1/2 of the CPU frequency.

There is a 24-bit watchdog timer and two 16-bit triple timer/counters, which exist on a system level. Here the system watchdog timer is clocked at 1/4 or 1/6 of the CPU frequency. The two triple timers/counters are always clocked at 1/4 or 1/6 of the CPU frequency, and are used to count the widths of signal pulses from an MIO pin or from the Programmable Logic.

4.2.6.1 The Private Timer

The Private timer is a decrementing 32-bit counter. The Timer Load Register contains the value copied to the counter. The counter decrements when the timer is enabled by using the timer enable bit. If a Cortex-A9 processor timer is in debug state, the counter only decrements when the Cortex-A9 processor returns to non debug state. When the counter reaches zero, the auto reload mode is enabled thus reloading the value in the Timer Load Register and then decrementing it from that value. . If auto reload mode is not enabled, the counter decrements down to zero and stops.

The Private timer block features an 8-bit value to qualify the clock period. It is moreover characterized by a configurable single-shot or auto-reload mode and configurable starting value for the counter.

4.3 The Programmable Logic

The Programmable Logic (PL) contributes in an affluent architecture which provides the user with the capability to configure it. The PL consists of configurable logic blocks, look-up tables with a 6-input and a memory capability within these look-up tables. It contains its own clock management with high-speed buffers and it routes for low-skew clock distribution. The clock management also features frequency synthesis, phase shifting, low-jitter clock generation and jitter filtering.

This Programmable logic is characterized by configurable Inputs and Output (I/Os). It has a High-performance Select-IO technology and high-frequency decoupling capacitors within the package so as to enhance the signal integrity. The configurable I/Os include digitally controlled impedance that can have 3 states for lowest power or the high-speed I/O operation.

Moreover, cascadable adders are contained in the Programmable Logic. Register and shift register functionality is included as well as digital signal processing.

4.3.1 The Advanced Microcontroller Bus Architecture (AMBA)

The Advanced Microcontroller Bus Architecture (AMBA) is an open-standard and on-chip interconnect specification used to connect and manage functional blocks in system on chip designs (SoC designs). The AMBA expedites the development on multiprocessor designs by using several controllers and peripherals. It is not only useful in micro-controller devices but also in SoC parts that include application processors that are used in portable mobile devices such as smart phones.

The first AMBA buses were Advanced System Bus (ASB) and Advanced Peripheral Bus (APB) introduced by ARM. The AMBA 2, the second version of the AMBA, has an added AMBA High-performance Bus (AHB) that is a single clock-edge protocol. ARM introduced the third generation, AMBA 3, including AXI to reach even higher performance interconnect and the Advanced Trace Bus (ATB) as part of the CoreSight on-chip debug and trace solution. The fourth version, AMBA 4, included the introduction of the AMBA 4 AXI4 which was then extended with AMBA 4 ACE. The fifth version AMBA 5 CHI (Coherent Hub Interface) specification was introduced with a re-designed high-speed transport layer and features designed to reduce congestion.

4.3.1.1 The Advanced Extensible Interface (AXI)

The Advanced Extensible Interface (AXI Interface) is the third generation of the AMBA interface that was defined in the AMBA 3 specification. It features a high performance and a high clock frequency. The AXI Interface is characterized by a separate address and data phase, support for unaligned data transfers using byte strobes, burst based transactions with only start address issued, issuing of multiple outstanding addresses with out of order responses and easy addition of register stages to provide timing closure. All of these features make the AXI Interface suitable of high speed sub-micrometer interconnect.

In this design the AXI does not support fixed burst type for the AXI ports into the DDRI, however it does support byte, half-word and word sub-width commands.

4.3.1.2 The AXI BRAM Controller

The AXI BRAM Controller is a soft IP core. This core is designed as an AXI endpoint slave IP for integration with the AXI interconnect and system master devices so as to communicate to local BRAM. The core supports both single and burst transactions to the BRAM and is optimized for performance.

4.3.1.3 The AXI GPIO

The AXI GPIO implements a general purpose input and output interface to the AXI interface. It is a 32-bit soft IP core that is designed to interface with the AXI4-Lite interface. These interfaces are connected directly to the ports of the master interconnect and the slave interconnect, without any additional FIFO buffering, therefore, the performance is constrained by the ports of the master interconnect and the slave interconnect. As a consequence, these interfaces are used for general-purpose and are not intended to achieve high performance.

The AXI GPIO features a standard AXI protocol, a 32-bit data bus width, a 12-bit master port ID width, a 6-bit slave port ID width, an 8 reads and 8 writes master port issuing capability and an 8 reads and 8 writes slave port acceptance capability. It supports configurable single or dual GPIO channels with a configurable channel width for GPIO pins ranging from 1bit to 32 bits. These GPIO bits can be dynamically programmed as an input or an output and each channel can be individually configured. The AXI GPIO subsidizes independent reset values for each bit in all of the registers and can generate optional interrupt requests.

4.3.2 The Random Access Memory (RAM)

The Random Access Memory (RAM) is a form of computer data storage. The RAM device allows data items to be read and written in approximately the same amount of time in which the data items were accessed. The time required to read and write data items may vary significantly depending on their physical locations on the recording medium, due to mechanical limitations such as media rotation speeds and arm movement delays.

Nowadays, the RAM takes the form of integrated circuits. It is normally associated with volatile types of memory where stored information is lost if power is removed, although many efforts have been made to develop non-volatile RAM chips. Other types of non-volatile memory exist that allow random access for read operations, but either do not allow write operations or have limitations on them. These include most types of ROM and a type of flash memory called NOR-Flash.

The RAM used in the Zynq design is a dual port 36 KB RAM with port widths ranging up to 72-bits wide. However, each block RAM can be divided into two completely independent dual 18 Kb block RAMs. It is of programmable FIFO logic and has a built in optional error correction circuitry.

Every Zynq-7000 AP SoC device has between 60 and 465 dual-port block RAMs, each storing 36 Kb. Each block RAM has two completely independent ports that share nothing but the stored data. Each memory access, whether read or write is controlled by the clock. All inputs, data, address, clock enables, and write enables are registered. The input address is always clocked, therefore retaining data until the next operation. An optional output data pipeline register allows higher clock rates at the cost of an extra cycle of latency. During a write operation, the data output can reflect either the previously stored data, the newly written data, or can remain unchanged.

Each port can be configured as 32K ×1, 16K ×2, 8K ×4, 4K ×9 (or x8), 2K ×18 (or x16), 1K ×36 (or 32), or 512 ×72 (or x64). The two ports can have different widths without any constraints. Each 64-bit-wide block RAM can generate, store, and utilize eight additional code bits and perform a single-bit error correction and double-bit error detection during the read process. This logic can also be used when writing to or reading from external 64- to 72-bit-wide memories.

5. Software Implementation Description.

5.1 Single Sobel and TEA Application

The main goal of these applications is to apply the Sobel filter to a digital image and then, by using the TEA algorithm, to encipher and decipher the processed image.

The Zynq-7000 provides two processor cores. The TEA algorithm runs on the processor "cortexa9_1" and the Sobel algorithm runs on the processor "cortexa9_0". The code for the Sobel filter is stored in and loaded from the DDR Memory (Double data rate synchronous dynamic random-access memory (DDR SDRAM)), and the TEA algorithm is stored in and loaded from RAM0 Memory (Random-access memory).

In order for the two applications to be able to communicate we use a pointer (*start), disclosed to both applications, that is located in the AXI BRAM Controller (the address is 0x40000000). This pointer acts as a flag and its inputs can either be 0 or 1. The TEA algorithm is the first to run but is stalled with a while loop, with the start pointer (flag) being its function argument. Unless the flag value remains the same, the application will never terminate. The purpose of this loop is to start the encryption and decryption after the image has been processed, namely after the Sobel application has terminated.

```
while (*start) {
                //waiting
        }

        *start =1;
```

Figure 5-0-1: While Loop In TEA Application

The Sobel application then follows. The digital image that we use here is a two-dimensional array with a size of 30x29 that we insert into a structure whose type is "image". The pointer "dstPoint" points to the array where the processed image is stored and will be used to insert the image into the DDR memory. For the image to be accessible to both applications the "dstPoint" pointer is stored in the DDR Memory (the address is 0x00200000). By using the instruction : *(dstPoint +(iter*2000) + count_sobel)= x->data[i][j] ; we store the processed pixels in the memory.

Proceeding to the main function, the image that we wish to process is stored in a structure whose type is "im". After the array has been filled, the Sobel function then takes place. When the function ends the result we receive is our processed image. The value of the pointer "*start" then changes. Since our image has been processed and the state of our flag has changed, we can then

proceed to the TEA algorithm.

Figure 5-0-2: Image After Sobel Filter

Figure 5-0-3: Original Image

The TEA algorithm is executed as follows. We insert into the one dimensional array in[] the 870 processed pixels of our image in order to apply the TEA mask. The array in_trans[2] is the input to the encryption algorithm. We enter the processed pixels stored in the DDR Memory in pairs because the algorithm can support up to 2 elements. Using the instruction: xtea_encipher (in_trans, out, key); we call the TEA encryption function. We then enter the results of the TEA mask, that works for two elements at a time, into the variables v0 and v1. After the encryption process is completed we insert, as an input, the enciphered pixels into the array in_trans[] in order to decrypt them. With the execution of the instructions: xtea_decipher (in_trans, out, key); the TEA decryption algorithm begins.

5.2 Multiple Sobel and TEA Application

The main purpose of running the Sobel and TEA applications multiple times is to study the effects on both processor cores by occupying them simultaneously.

To calculate the estimated time each application takes to finish we used the functions that activate the Xilinx hardware timer (SCU Timer). More specifically to start the timer we execute the instruction: XscuTimer_Start(&Timer); and to stop it with the instruction: XScuTimer_Stop(&Timer);. The next stage was to extract the counted value with the instruction: Cnt=XScuTimer_GetCounterValue(&Timer);. The timer counts in reverse, so, as to calculate the real time an application takes to complete itself, we need to subtract the counters value with the timers initial load value.

The measurements that were taken showed that the TEA application took a significantly longer time to finish than the Sobel application. To achieve simultaneous occupation on the two processor cores the Sobel applications runs three times and the TEA application two times. The same process as before is implemented with the only difference of a “for” loop in both applications to achieve repetition.

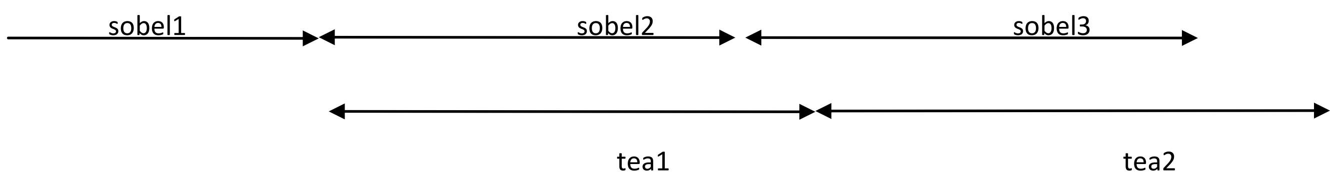


Figure 5-0-4: Process of Multiple Sobel and TEA Applications

5.2 Performance Monitoring Unit Application

The main purpose of this application (source code) is to count certain events using the PMU event counters. The files v7_pmu.S and v7_pmu.h, used in our thesis, were originally provided by Arm but modified so as to be GCC friendly. The assembly code file access the unit by using the CP15 interface. The application is implemented in three main parts: enabling, counting and disabling.

At first we enable the Performance Monitoring Unit. On other processors (for example ARM11) the counters start immediately, but on the ARM Cortex-a9 each one of the six event counters need to be individually enabled. The parameters for this function are the numbers 0 to 5 (each number for the corresponding event counter). We also enable the cycle counter register. After the enabling we reset the programmable counters and the cycle counter register to its original value.

```
void enable_pmu(void);

void enable_ccnt(void);

void enable_pmn(unsigned int
counter);
```

Figure 5-0-5: Enabling Functions

In order to be able to conduct the measurements we have to configure the event counters for the specific events. The function used is the void pmn_config(), that sets the event for the programmable counter to record. The attributes of this function are both the counter which is programmed and the event code. Before the main function, of each application, runs, we read the value of the counters. We also read the value of the cycle counter register. The value returned is assigned to a position in an integer table. The same procedure is repeated after the main function of each application has run. Basically, we want the measure the impact this function (that is part of the applications we have described previously) has on the system.

```
void pmn_config(unsigned int counter,
unsigned int event);

unsigned int read_ccnt(void);

unsigned int read_pmn(unsigned int
counter);
```

Figure 5-0-6: Counting Functions

After we have read the values of the counters, set to record certain events, we disable both the cycle counter register and the Performance Monitoring Unit with the functions void disable_ccnt(void) and void disable_pmu(void) correspondingly. The values after each event is

counted are stored in a table. The subtraction between the result of the counter before and after the main application function is the actual and final value of the event.

```
void disable_pmu(void);

void disable_ccnt(void);
```

Figure 5-0-7: Disabling Functions

This source code is inserted into the Sobel, TEA and Black Scholes applications. Specifically, in the Sobel application, the Performance Monitoring Unit calculates the results of the events before and after the function sobel(im,r) . In the TEA application, the PMU calculates the results before and after the ecryption and decryption. Finally, in the Black Scholes application, the results calculated by the PMU are before and after each of the five main functions it contains (Asset path, Binomial, BSF, Forward and MC).

6. Measurements-Results

The Performance Monitoring Unit was applied to applications in order to study the effects they have on a processing system. The applications used are Sobel, Tea and Black Scholes. In the case of Sobel combined with Tea, measurements were taken when they run once and then again when they run multiple times. Each application is studied with the usage of cache and without it. The bandwidths presented below are compared to the overall cycle counts.

As we have previously mentioned, to achieve simultaneous occupation on the two processor cores the Sobel applications runs three times and the TEA application two times. In reference to the TEA applications, we can observe that the PMU gives us no results for the first application. This is because the PMU cannot be used by both processors simultaneously.

The number of cycles counted by the PMU for the completion of the first Sobel is almost the same as when we run the Sobel application by itself. The cycle count for the following Sobel applications is slightly elevated. This is due to the fact that both processors work simultaneously and there is a certain strain of the processing system. . The cycle count for the TEA application is almost the same as when it runs on its own. This is logical since the total of the Sobel applications have terminated before the last TEA application ends; therefore there is no particular strain on the system.

The cycles, of the following measurements, are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

6.1Formulas

- The Cycle Counter Register (ccnt) provided by the PMU counts the same thing as the event Cycle Count (0x11). The equation linking these two is: cycle count/64=ccnt.
- The ScuTimer used is clocked at ½ of the CPUs frequency
- The Cortex-A9 CPU Clock Frequency is 666 MHz.
- The events Data read (0x06) and Data writes (0x07) added together equal the event Load/Store Instructions (0x72).
- The cache ratio is the Data cache misses (0x03) divided by the Data cache access (0x04).

6.2Sobel and TEA Applications

6.2.1Sobel and TEA Application Time

As we have previously mentioned, we can see that the TEA application takes more time to complete than Sobel. There is also a severe difference between the time each application needs to terminate with and without the use of caches (Chart 1). The values of the cycle count for Sobel and TEA with cache are less than the cycle count without caches. The explanation to this difference is that with the use of a cache, data can be read from it rather than reading data from a slower data source, thus making the system perform faster. The difference between the Sobel application with and without cache is 1032619 cycles whilst for the Tea application 2022721 cycles.

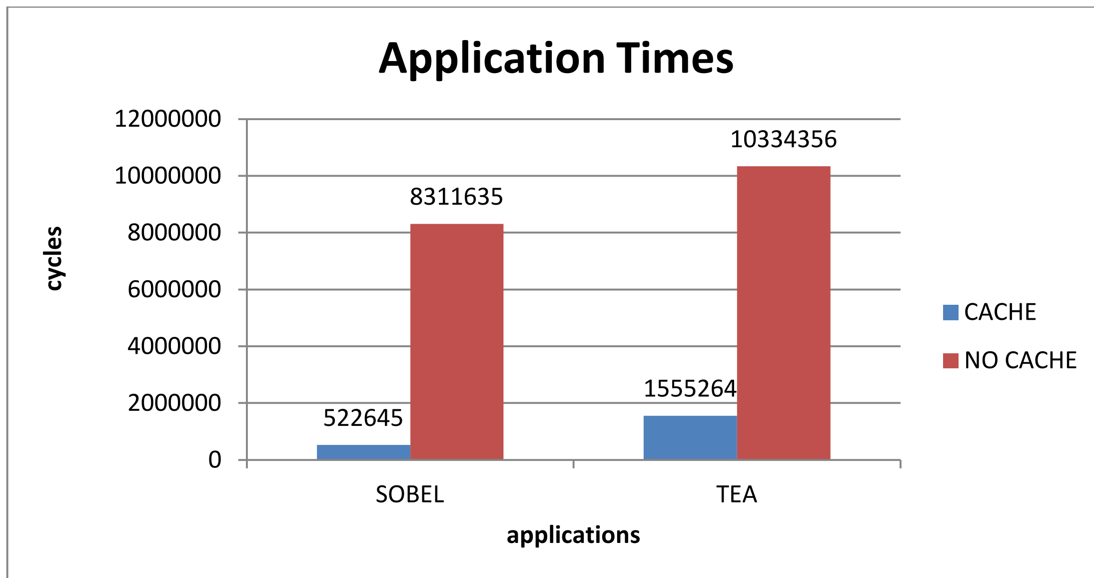


Chart 1 : Sobel and TEA Application Times [1]

6.2.2 Sobel and TEA Instruction Cache Miss Bandwidths

The Instruction Cache Miss (0x01) for the single Sobel and TEA application has the same value with and without the use of the cache memories. As the applications proceed to run the values decrease. In the multiple versions of the Sobel applications the value is always zero.

[1] The cycles are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

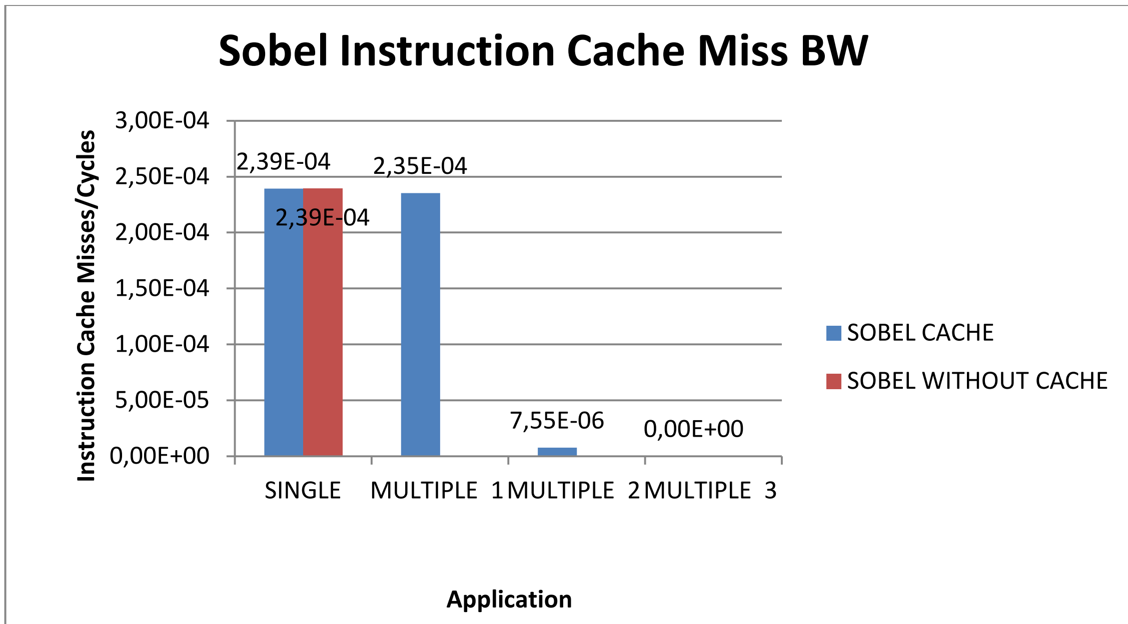


Chart 2: The Instruction Cache Miss Bandwidths for the Sobel Application[2]

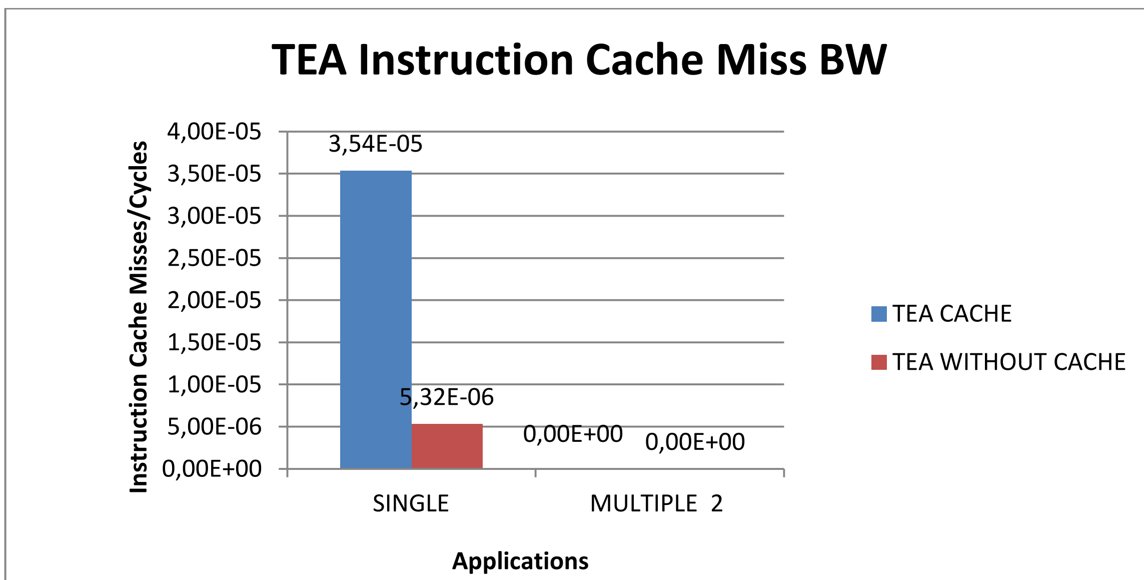


Chart 3: The Instruction Cache Miss Bandwidths for the TEA Application

6.2.3 Sobel and TEA Data Cache Miss Bandwidths

The single Sobel and TEA applications manifest no result when they run without the use of the cache memory. This is normal since no cache is used. The Data Cache Misses for the multiple versions of the Sobel and TEA applications with and without the use of cache are always zero.

[2] The cycles for both charts depicted are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

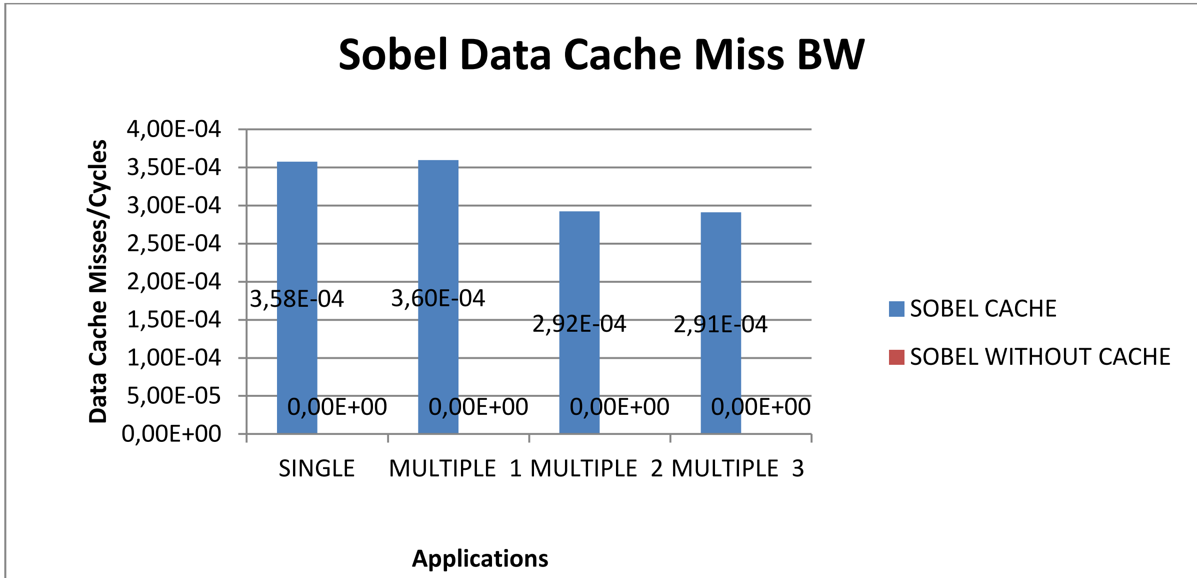


Chart 4: The Data Cache Miss Bandwidths for the Sobel Application[3]

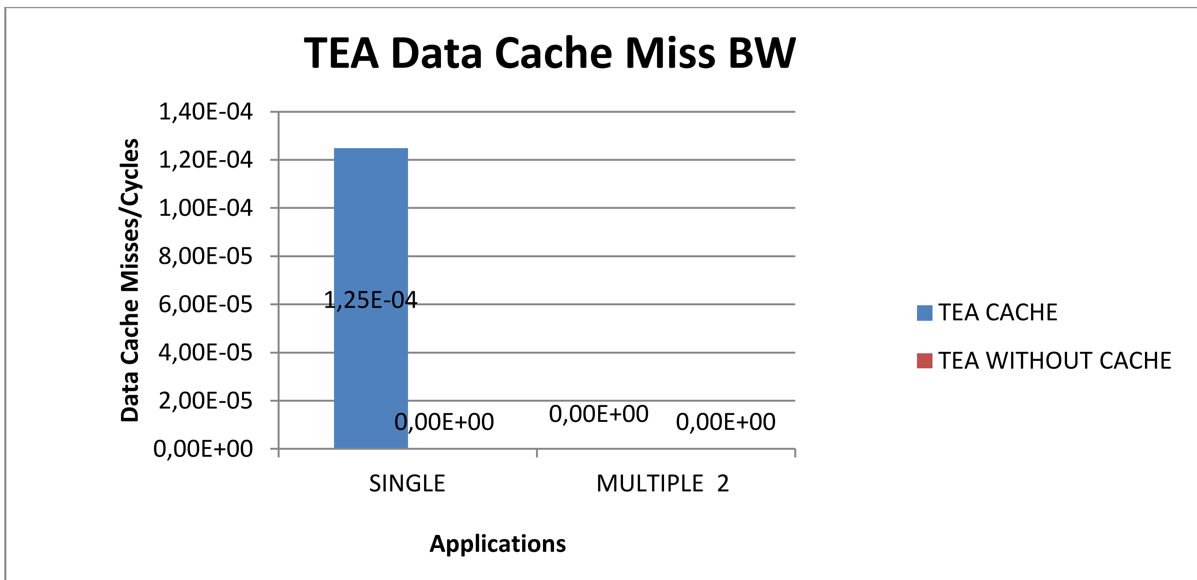


Chart 5: The Data Cache Miss Bandwidths for the TEA Application

[3] The cycles for both charts depicted are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

6.2.4 Sobel and TEA Data Cache Access Bandwidths

The Data Cache Access for both the Sobel and TEA applications that employ the cache memory maintains similar values and has minor variations. However there seems to be a data cache access when the cache is disabled. This could be because, the Performance Monitoring Unit counts any attempt to access data.

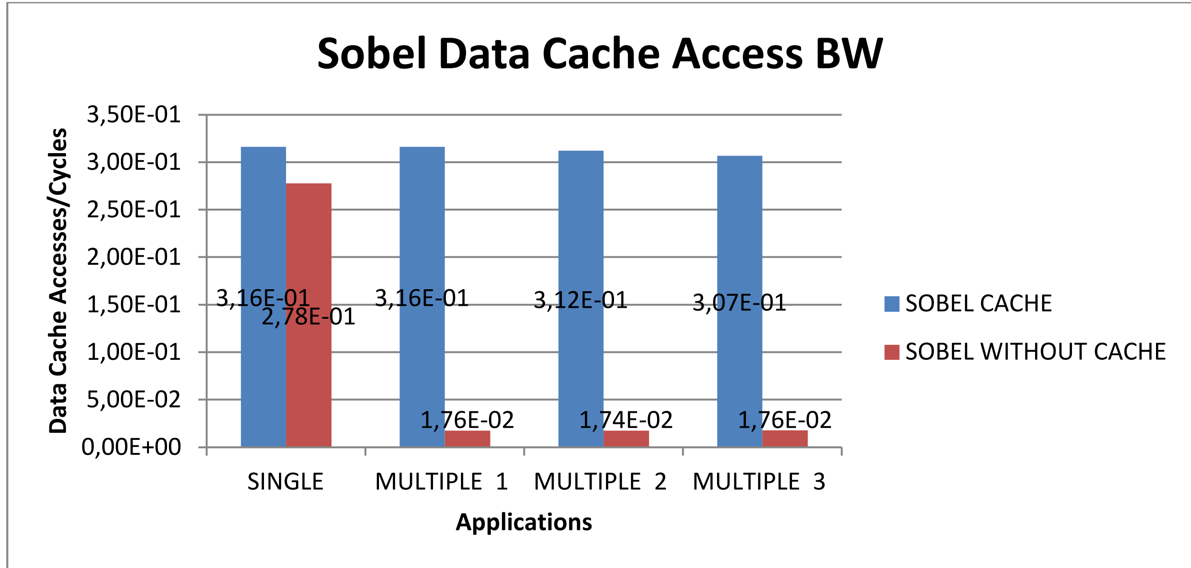


Chart 6: The Data Cache Access Bandwidths for the Sobel Application[4]

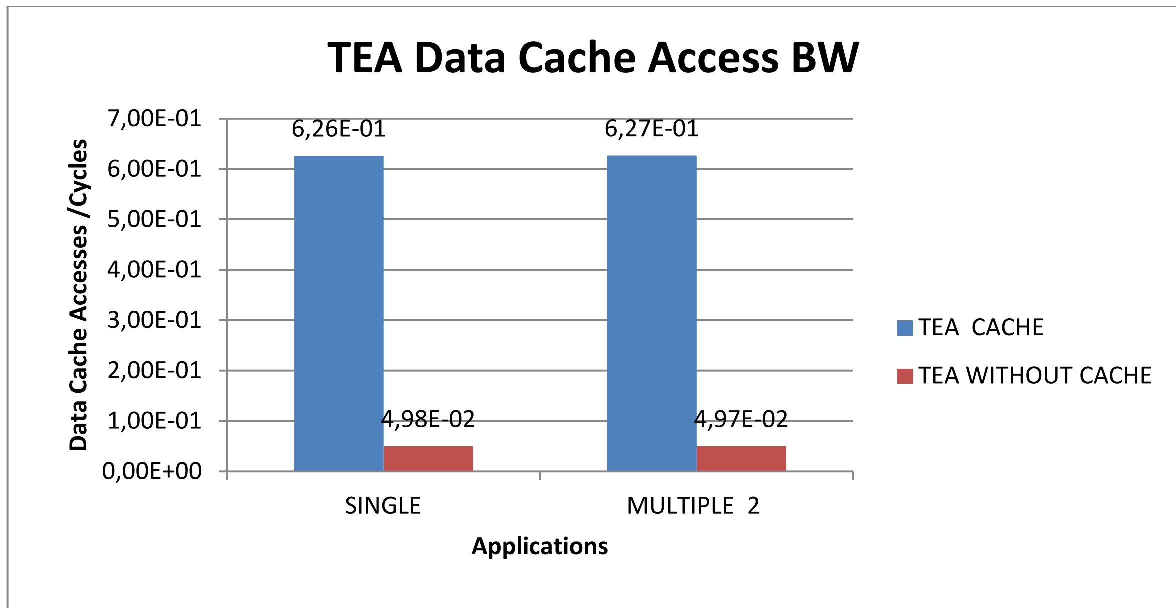


Chart 7: The Data Cache Access Bandwidths for the TEA Application

[4] The cycles for both charts depicted are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

6.2.5 Sobel and TEA Load/Store Instructions Bandwidths

The event Load/Store Instructions (0x72) which is the overall memory access is in the same range in both cases with or without the usage of the cache memory in the single Sobel application. The same scenario applies to the single TEA application. In the cases of the multiple Sobel and TEA applications with and without the utilization of the cache, the values are similar and have small variations.

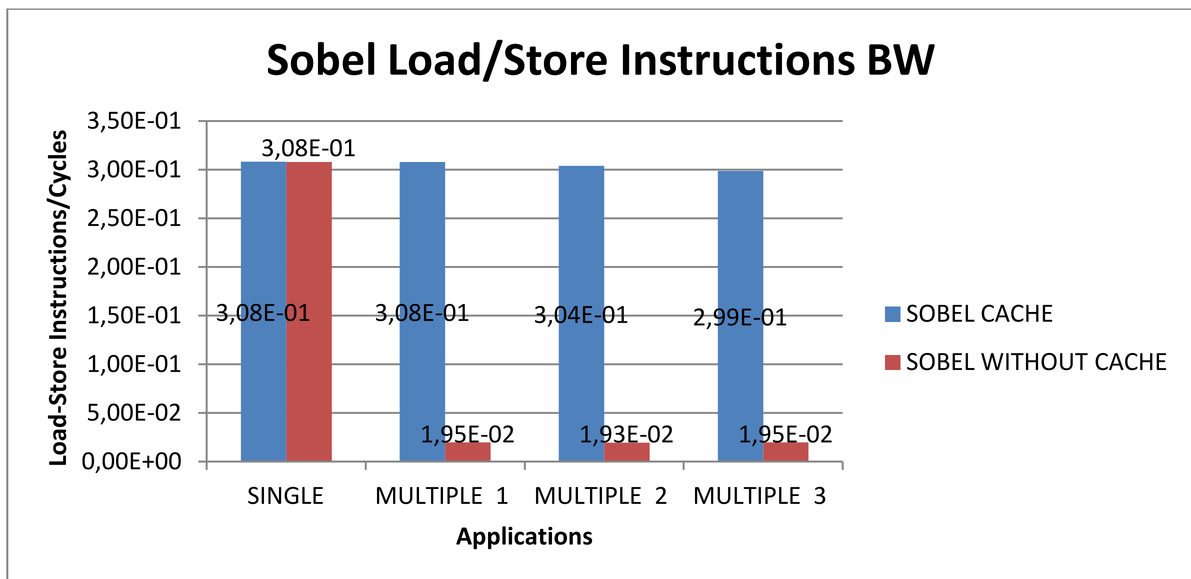


Chart 8: The Load/Store Instructions Bandwidths for the Sobel Application[5]

[5] The cycles for both charts depicted are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

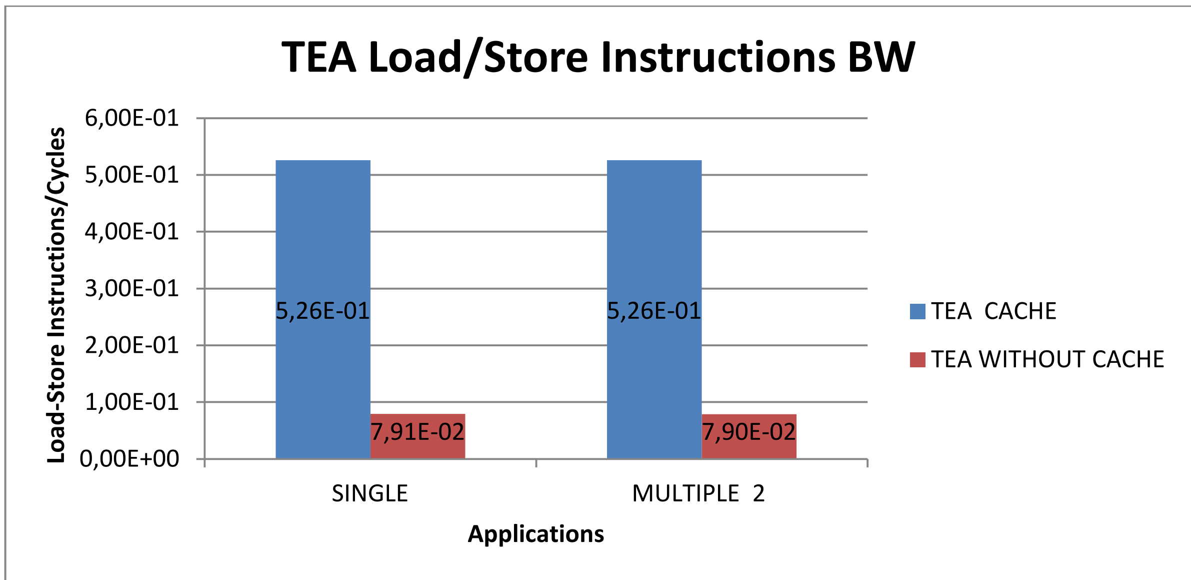


Chart 9: The Load/Store Instructions Bandwidth for the TEA Application[6]

6.2.6 Sobel and TEA Cache Ratio

The Cache Ratio for the Sobel and TEA applications (either single or multiple) without cache is always zero. This circumstance is normal since there is no cache utilization. On the other hand, when the cache is wielded the cache ratio values gradually decrease.

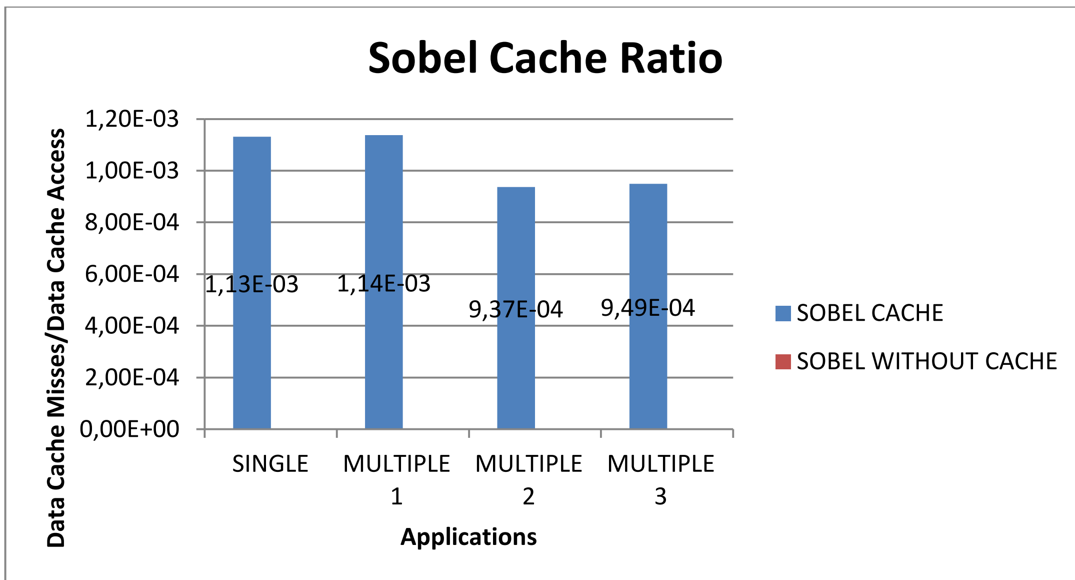


Chart 10: The Cache Ratio for the Sobel Application

[6] The cycles for both charts depicted are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

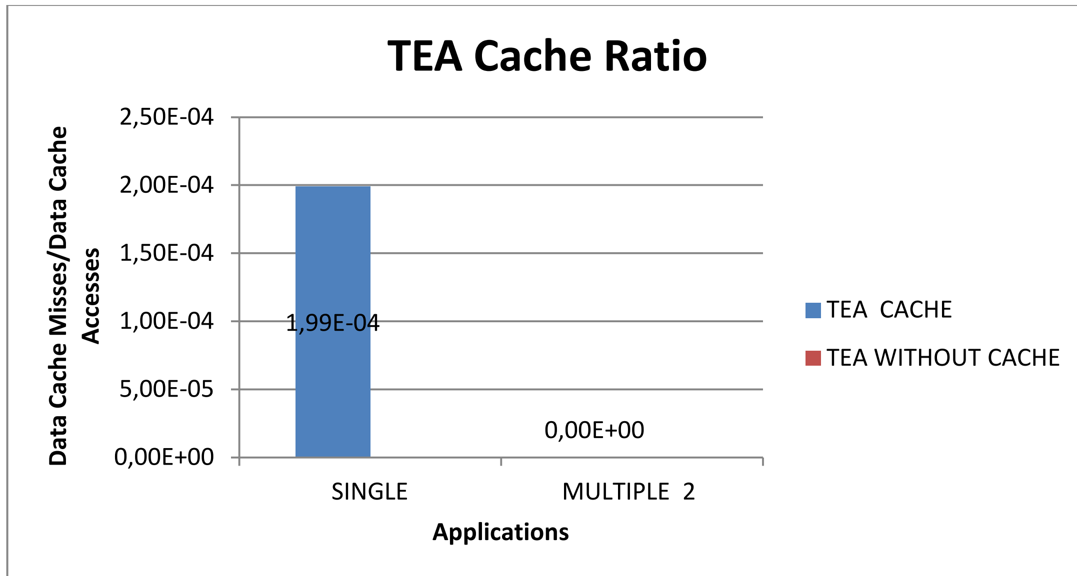


Chart 11: The Cache Ratio for the TEA Application[7]

6.3Black Scholes Application

Studies were made on the Black Scholes benchmark by using all five of its applications and testing the impact these applications have on the processing system with the use of caches and without them. Generally, the conclusions made are similar to those made concerning the Sobel and TEA applications. All the cycle counts are significantly greater without the use of cache as opposed to with cache, which is expected since in this case we have to access a slower data source. The value of the event Data cache miss without cache for all five of the tests is always zero. The event Data cache access without cache always has a value.

The apparently big cycle count regarding the Mc Test is due to its vast number of sample. Additional tests were done based on the number of samples. As the samples increases, so does the cycle count, which is quite obvious since the more the samples, the more the processes made by the system.

The cycles, of the following measurements, are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

[7] The cycles are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

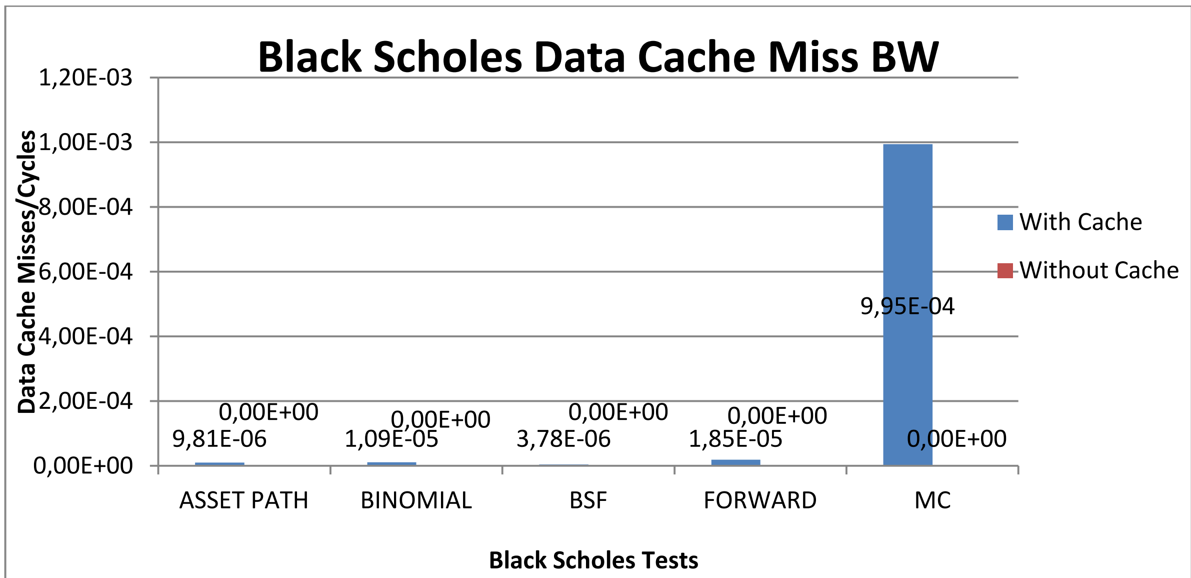


Chart 12: The Data Cache Miss Bandwidth for all five of the Black Scholes Applications [8]

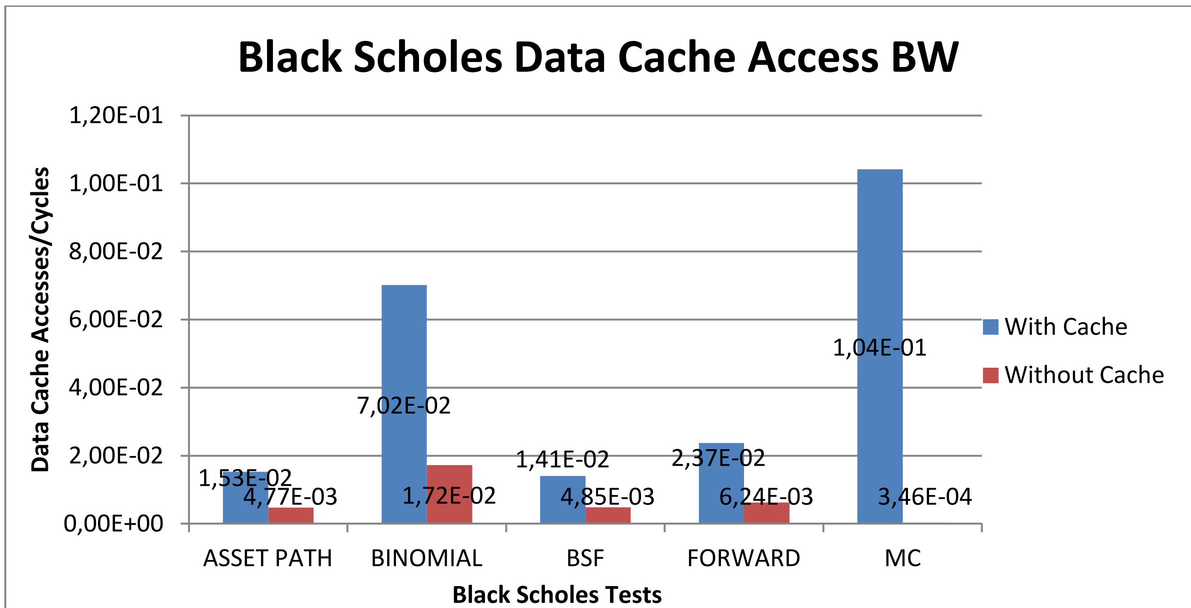


Chart 13: The Data Cache Access Bandwidth for all five of the Black Scholes Applications

[8] The cycles for both charts depicted are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

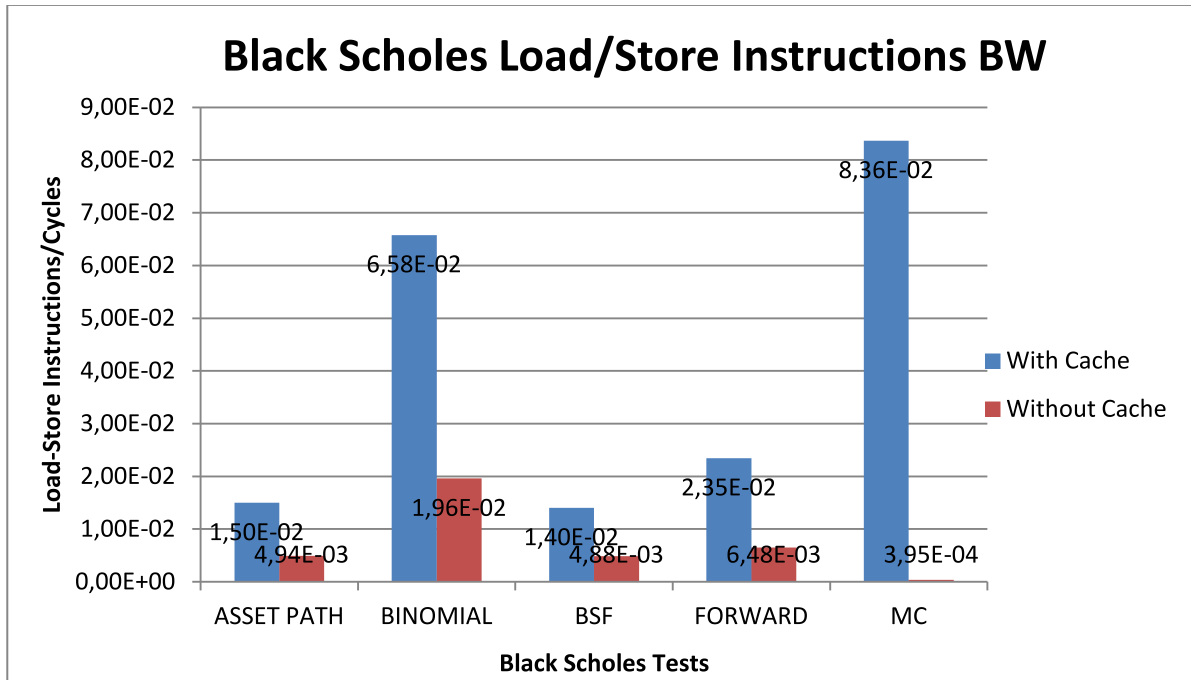


Chart 14: The Load/Store Instructions Bandwidth for all five of the Black Scholes Applications[9]

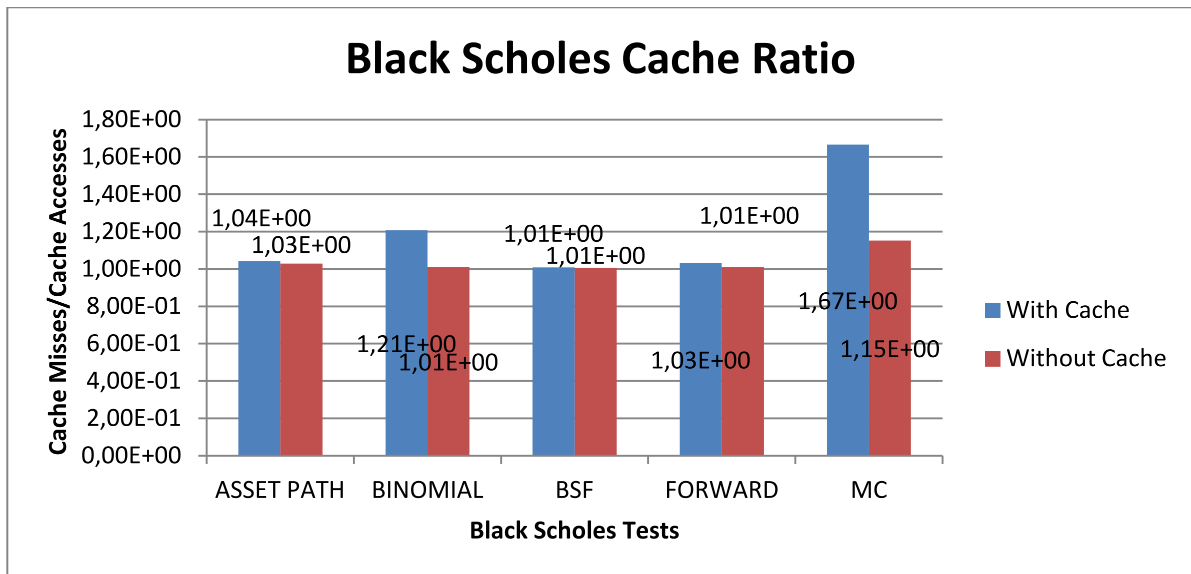


Chart 15: The Cache Ratio for all five of the Black Scholes Applications

[9] The cycles for both charts depicted are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

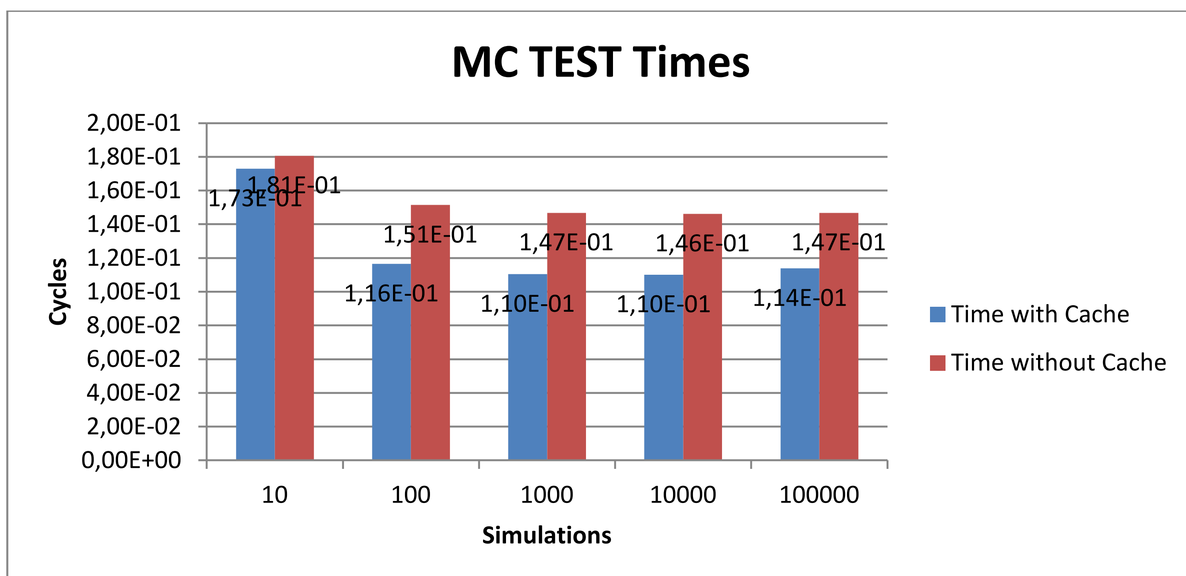


Chart 166: The Times For the MC Test[10]

[10] The cycles are calculated based on the Cortex-A9 CPU Clock Frequency, which is 666 MHz.

7. Conclusions and Future Work

After studying many of the trials that we carried out, we can now make certain observations about the use of Performance Monitoring Units in applications.

When an application has disabled its cache, the access to data is done through slower and inflexible data sources thus delaying the system in its whole. On the other hand, high Cycle Count Rate can be due to the fact that both processors are working simultaneously. When the processors are required to work at the same time, a certain strain is applied to the system causing more processes to use the same resources at the same time, thus increasing the Cycle Count.

We are now able to understand and comprehend the consumption in our systems resources through the use of the Performance Monitoring Unit, therefore we can use this information to our advantage and guide the processors to perform quality of service.

7.1 Future Work

The Performance Monitoring Unit has given us an idea as to how the system functions and handles its resources.

We intend to focus on developing a system that will provide real-time quality of service. By using the Performance Monitoring Unit, the one core will run various applications and the other core will act as a monitor.

This can be expanded on to Linux Operating Systems, where this monitor will temporarily pause processes that have a high resource occupation in order to let other applications run.

Bibliography

[1] "CASHIER: A Cache Energy Saving Technique for QoS Systems".

[2] "Energy Efficient Scheduling of Real-Time".

[3] Dionisios Pnevmatikatos George Kornaros, "Real-Time Monitoring of Multicore SoCs Through Specialized Hardware Agents on NoC Network Interfaces".

[4] Younghyun Kim, Sangyoung Park and Naehyuck Chang Youngjin Cho, "SystemLevel Power Estimation using an OnChip".

[5] Anup Buchke, Dr. Yann-Hang Lee Aman Singh, "A Study of Performance Monitoring Unit, Perf and Perf-Events Subsystem".

[6] *ARM Architecture Reference Manual*.

[7] *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*.

[8] *Cortex-A9 Technical Reference Manual*.

[9] Steven Gerding, "The Extreme Benchmark Suite: Measuring High- Performance Embedded Systems".

[10] Kevin Castor, "Hardware Testing and Benchmarking Methodology".

[11] Irwin Sobel, "History and Definition of the Sobel Operator".

[12] Matthew D. Russell. Tinyness, "An Overview of TEA and Related Ciphers. Draft".

[13] Matthew D. Russell, "Tinyness: An Overview of TEA and Related Ciphers".

[14] *http://parsec.cs.princeton.edu/overview.htm*.

[15] *Zynq-7000 AP SoC Technical Reference Manual*.

Appendix A

1.1 Additional Performance Monitoring Events

1.1.1 Implemented architectural events

Number	Event
0x00	Software increment
0x01	Instruction cache miss
0x02	Instruction micro TLB miss
0x03	Data cache miss
0x04	Data cache access
0x05	Data micro TLB miss
0x06	Data read
0x07	Data writes
0x09	Exception taken
0x0A	Exception return
0x0B	Write context ID
0x0C	Software change of the PC
0x0D	Immediate branch
0x0F	Unaligned load or store
0x10	Branch mispredicted or not predicted
0x11	Cycle count
0x12	Predictable branches

1.1.2 Coretex-A9 Specific Events

Event	Description	Value
0x40	Java bytecode execute	Approximate
0x41	Software Java bytecode executed	Approximate
0x42	Jazelle backward branches executed	Approximate
0x50	Coherent linefill miss	Precise
0x51	Coherent linefill hit	Precise
0x60	Instruction cache dependent stall cycles	Approximate
0x61	Data cache dependent stall cycles	Approximate
0x62	Main TLB miss stall cycles	Approximate
0x63	STREX passed	Precise
0x64	STREX failed	Precise
0x65	Data eviction	Precise
0x66	Issue does not dispatch any instruction	Precise
0x67	Issue is empty	Precise
0x68	Instructions coming out of the core renaming stage	Approximate
0x69	Number of data linefills	Precise
0x6A	Number of prefetcher linefills	Precise
0x6B	Number of hits in prefetched cache lines	Precise
0x6E	Predictable function returns	Approximate
0x70	Main execution unit instructions	Approximate
0x71	Second execution unit instructions	Approximate
0x72	Load/Store Instructions	Approximate
0x73	Floating-point instructions	Approximate
0x74	NEON instructions	Approximate
0x80	Processor stalls because of PLDs	Approximate
0x81	Processor stalled because of a write to memory	Approximate
0x82	Processor stalled because of instruction side main TLB miss	Approximate
0x83	Processor stalled because of data side main TLB miss	Approximate
0x84	Processor stalled because of instruction micro TLB miss	Approximate
0x85	Processor stalled because of data micro TLB miss	Approximate
0x86	Processor stalled because of DMB	Approximate
0x8A	Integer clock enabled	Approximate
0x8B	Data engine clock enabled	Approximate
0x8C	NEON SIMD clock enabled	Approximate
0x8D	Instruction TLB allocation	Approximate
0x8E	Data TLB allocation	Approximate
0x90	ISB instructions	Precise
0x91	DSB instructions	Precise

0x92	DMB instructions	Approximate
0x93	External interrupts	Approximate
0xA0	PLE cache line request completed	Precise
0xA1	PLE cache line request skipped	Precise
0xA2	PLE FIFO flush	Precise
0xA3	PLE request completed	Precise
0xA4	PLE FIFO overflow	Precise
0xA5	PLE request programmed	Precise

[11]

1.2 PMU Assembly Access Functions

- unsigned int getPMN(void);

```
getPMN:
  MRC     p15, 0, r0, c9, c12, 0 /* Read PMNC Register  */
  MOV     r0, r0, LSR #11        /* Shift N field down to bit 0 */
  AND     r0, r0, #0x1F          /* Mask to leave just the 5 N bits     */
  BX      lr
```

- void pmn_config(unsigned int counter, unsigned int event);

```
  pmn_config:
AND     r0, r0, #0x1F          /* Mask to leave only bits 4:0 */
MCR     p15, 0, r0, c9, c12, 5 /* Write PMNXSEL Register      */
MCR     p15, 0, r1, c9, c13, 1 /* Write EVTSELx Register      */
BX      lr
```

- void ccnt_divider(int divider);

```
ccnt_divider:
MRC     p15, 0, r1, c9, c12, 0  /* Read PMNC  */

 CMP     r0, #0x0               /* IF (r0 == 0)       */
BICEQ   r1, r1, #0x08           /* THEN: Clear the D bit (disables the divisor)       */
ORRNE   r1, r1, #0x08           /* ELSE: Set the D bit (enables the divisor)  */

MCR     p15, 0, r1, c9, c12, 0  /* Write PMNC */
BX      lr
```

[11] For more information regarding the Cortex-A9 Performance Monitoring Events please refer to the Cortex-A9 Technical Reference Manual Revision: r4p1.

- void enable_pmu(void);

```
enable_pmu:
  MRC    p15, 0, r0, c9, c12, 0  /* Read PMNC  */
  ORR    r0, r0, #0x01           /* Set E bit  */
  MCR    p15, 0, r0, c9, c12, 0  /* Write PMNC */
  BX     lr
```

- void disable_pmu(void);

```
disable_pmu:
  MRC    p15, 0, r0, c9, c12, 0  /* Read PMNC  */
  BIC    r0, r0, #0x01           /* Clear E bit        */
  MCR    p15, 0, r0, c9, c12, 0  /* Write PMNC */
  BX     lr
```

- void enable_ccnt(void);

```
enable_ccnt:
  MOV    r0, #0x80000000         /* Set C bit  */
  MCR    p15, 0, r0, c9, c12, 1  /* Write CNTENS Register      */
  BX     lr
```

- void disable_ccnt(void);

```
disable_ccnt:
  MOV    r0, #0x80000000         /* Clear C bit        */
  MCR    p15, 0, r0, c9, c12, 2  /* Write CNTENC Register      */
  BX     lr
```

- void enable_pmn(unsigned int counter);

```
enable_pmn:
  MOV    r1, #0x1                /* Use arg (r0) to set which counter to disable       */
  MOV    r1, r1, LSL r0

  MCR    p15, 0, r1, c9, c12, 1  /* Write CNTENS Register      */
  BX     lr
```

- void disable_pmn(unsigned int counter);

```
disable_pmn:
  MOV    r1, #0x1                /* Use arg (r0) to set which counter to disable       */
  MOV    r1, r1, LSL r0
```

```
  MCR     p15, 0, r1, c9, c12, 1  /* Write CNTENS Register      */
  BX      lr
```

- unsigned int read_ccnt(void);

```
read_ccnt:
  MRC     p15, 0, r0, c9, c13, 0 /* Read CCNT Register  */
  BX      lr
```

- unsigned int read_pmn(unsigned int counter);

```
read_pmn:
  AND     r0, r0, #0x1F          /* Mask to leave only bits 4:0 */
  MCR     p15, 0, r0, c9, c12, 5 /* Write PMNXSEL Register      */
  MRC     p15, 0, r0, c9, c13, 2 /* Read current PMNx Register  */
  BX      lr
```

- unsigned int read_flags(void);

```
read_flags:
  MRC     p15, 0, r0, c9, c12, 3 /* Read FLAG Register  */
  BX      lr
```

- void write_flags(unsigned int flags);

```
write_flags:
  MCR     p15, 0, r0, c9, c12, 3 /* Write FLAG Register */
  BX      lr
```

- void enable_ccnt_irq(void);

```
enable_ccnt_irq:
  MOV     r0, #0x80000000
  MCR     p15, 0, r0, c9, c14, 1  /* Write INTENS Register      */
  BX      lr
```

- void disable_ccnt_irq(void);

```
disable_ccnt_irq:
  MOV     r0, #0x80000000
```

```
MCR    p15, 0, r0, c9, c14, 2   /* Write INTENC Register     */
BX     lr
```

- void disable_pmn_irq(unsigned int counter);

```
disable_pmn_irq:
 MOV    r1, #0x1              /* Use arg (r0) to set which counter to disable       */
 MOV    r0, r1, LSL r0
 MCR    p15, 0, r0, c9, c14, 2  /* Write INTENC Register      */
 BX     lr
```

- void reset_pmn(void);

```
reset_pmn:
 MRC    p15, 0, r0, c9, c12, 0  /* Read PMNC  */
 ORR    r0, r0, #0x02          /* Set P bit (Event Counter Reset)    */
 MCR    p15, 0, r0, c9, c12, 0  /* Write PMNC */
 BX     lr
```

- void reset_ccnt(void);

```
reset_ccnt:
 MRC    p15, 0, r0, c9, c12, 0  /* Read PMNC  */
 ORR    r0, r0, #0x04          /* Set C bit (Event Counter Reset)    */
 MCR    p15, 0, r0, c9, c12, 0  /* Write PMNC */
 BX     lr
```

- void pmu_software_increment(unsigned int counter);

```
pmu_software_increment:
 MOV    r1, #0x01
 MOV                r1, r1, LSL r0
 MCR    p15, 0, r1, c9, c12, 4 /* Write SWINCR Register       */
 BX     lr
```

Appendix B

1.1 Additional Measurements

1.1.1 Single Sobel and TEA Application

EVENT		L1 Disabled	L1 Disabled DEC
0x01	Instruction cache miss	0	0
0x03	Data cache miss	0	0
0x04	Data cache access	236C6	145094
0x11	Cycle count	2C52C4	2904772
0x72	Load/Store Instructions	274D7	160983

Table 1: Single Sobel With L1 Cache Disabled

EVENT		L1 Disabled	L1 Disabled DEC
0x01	Instruction cache miss	0	0
0x03	Data cache miss	0	0
0x04	Data cache access	7D94E	514382
0x11	Cycle count	9D1FC8	10297288
0x72	Load/Store Instructions	C7B0C	817932

Table 2: Single TEA With L1 Cache Disabled

EVENT		L2 Disabled	L2 Disabled DEC
0x01	Instruction cache miss	80	128
0x03	Data cache miss	BC	188
0x04	Data cache access	2850F	165135
0x11	Cycle count	7F8A3	522403
0x72	Load/Store Instructions	2749C	160924

Table 3: Single Sobel with L2 Cache Disabled

[12]

[12] The measurements are featured in Hexadecimal and Decimal format

EVENT		L2 Disabled	L2 Disabled DEC
0x01	Instruction cache miss	80	128
0x03	Data cache miss	BC	188
0x04	Data cache access	2850F	165135
0x11	Cycle count	7F8A3	522403
0x72	Load/Store Instructions	2749C	160924

Table 4: Single TEA With L2 Cache Disabled

EVENT		I CACHE Disabled	I CACHE Disabled DEC
0x01	Instruction cache miss	0	0
0x03	Data cache miss	BD	189
0x04	Data cache access	282BC	164540
0x11	Cycle count	3B18D5	3872981
0x72	Load/Store Instructions	26FEF	159727

Table 5: Single Sobel With I cache Disabled

EVENT		I CACHE Disabled	I CACHE Disabled DEC
0x01	Instruction cache miss	0	0
0x03	Data cache miss	C3	195
0x04	Data cache access	E069A	919194
0x11	Cycle count	36ADB9	3583417
0x72	Load/Store Instructions	C6BA4	813988

Table 6: Single TEA With I Cache Disabled

EVENT		D CACHE Disabled	D CACHE Disabled DEC
0x01	Instruction cache miss	83	131
0x03	Data cache miss	0	0
0x04	Data cache access	236D1	145105
0x11	Cycle count	64A6BA	6596282
0x72	Load/Store Instructions	274D8	160984

Table 7: Single Sobel With D Cache Disabled

EVENT		D CACHE Disabled	D CACHE Disabled DEC
0x01	Instruction cache miss	3E	62
0x03	Data cache miss	0	0
0x04	Data cache access	7D951	514385
0x11	Cycle count	8DF389	9302921
0x72	Load/Store Instructions	C7B13	817939

Table 8: Single TEA With D Cache Disabled

1.1.2 *Multiple Sobel and TEA Application*

EVENT		Sobel 1	EVENT		Sobel 2	EVENT		Sobel 3
0x01	Instruction cache miss	0	0x01	Instruction cache miss	0	0x01	Instruction cache miss	0
0x03	Data cache miss	0	0x03	Data cache miss	0	0x03	Data cache miss	0
0x04	Data cache access	236CD	0x04	Data cache access	236C5	0x04	Data cache access	236CD
0x11	Cycle count	2C18BD	0x11	Cycle count	2C1939	0x11	Cycle count	2CECA0
0x72	Load/Store Instructions	274E6	0x72	Load/Store Instructions	274D2	0x72	Load/Store Instructions	27529

EVENT		TEA 1		EVENT		TEA 2
0x01	Instruction cache miss	FF		0x01	Instruction cache miss	0
0x03	Data cache miss	FF		0x03	Data cache miss	0
0x04	Data cache access	FF		0x04	Data cache access	7D9B6
0x11	Cycle count	FF		0x11	Cycle count	9D7D44
0x72	Load/Store Instructions	FF		0x72	Load/Store Instructions	C7B7F

Table 9: Multiple Sobel and TEA With L1 Cache Disabled[13]

EVENT		Sobel 1	EVENT		Sobel 2	EVENT		Sobel 3
0x01	Instruction cache miss	7F	0x01	Instruction cache miss	4	0x01	Instruction cache miss	0
0x03	Data cache miss	0	0x03	Data cache miss	0	0x03	Data cache miss	0
0x04	Data cache access	236CE	0x04	Data cache access	236E7	0x04	Data cache access	236D8
0x11	Cycle count	64079C	0x11	Cycle count	6415CE	0x11	Cycle count	646AED
0x72	Load/Store Instructions	274DF	0x72	Load/Store Instructions	274F1	0x72	Load/Store Instructions	2753A

EVENT		TEA 1		EVENT		TEA 2
0x01	Instruction cache miss	FF		0x01	Instruction cache miss	0
0x03	Data cache miss	FF		0x03	Data cache miss	0
0x04	Data cache access	FF		0x04	Data cache access	7D940
0x11	Cycle count	FF		0x11	Cycle count	8E2CDB
0x72	Load/Store Instructions	FF		0x72	Load/Store Instructions	C7AE6

Table 10: Multiple Sobel and TEA With D Cache Disabled

[13] The measurements are featured in Hexadecimal format

Sobel 1	EVENT		Sobel 2	EVENT		Sobel 3	
0	0x01	Instruction cache miss	0	0x01	Instruction cache miss	0	
BB	0x03	Data cache miss	99	0x03	Data cache miss	9D	
282B3	0x04	Data cache access	282C5	0x04	Data cache access	282BA	
3BF510	0x11	Cycle count	3CF258	0x11	Cycle count	3F8051	
26FE3	0x72	Load/Store Instructions	26FB7	0x72	Load/Store Instructions	26FC8	
	EVENT		TEA 1		EVENT		TEA 2
	0x01	Instruction cache miss	FF		0x01	Instruction cache miss	0
	0x03	Data cache miss	FF		0x03	Data cache miss	0
	0x04	Data cache access	FF		0x04	Data cache access	D650E
	0x11	Cycle count	FF		0x11	Cycle count	36ACBE
	0x72	Load/Store Instructions	FF		0x72	Load/Store Instructions	C6B81

Table 11: Multiple Sobel and TEA With I Cache Disabled

EVENT		Sobel 1	EVENT		Sobel 2	EVENT		Sobel 3	
0x01	Instruction cache miss	77	0x01	Instruction cache miss	4	0x01	Instruction cache miss	0	
0x03	Data cache miss	4F	0x03	Data cache miss	99	0x03	Data cache miss	9B	
0x04	Data cache access	2826A	0x04	Data cache access	285A3	0x04	Data cache access	28597	
0x11	Cycle count	7F395	0x11	Cycle count	80C6E	0x11	Cycle count	83260	
0x72	Load/Store Instructions	2747E	0x72	Load/Store Instructions	2747E	0x72	Load/Store Instructions	274B5	
			EVENT		TEA 1		EVENT		TEA 2
			0x01	Instruction cache miss	FF		0x01	Instruction cache miss	0
			0x03	Data cache miss	FF		0x03	Data cache miss	0
			0x04	Data cache access	FF		0x04	Data cache access	E60B7
			0x11	Cycle count	FF		0x11	Cycle count	17B90E
			0x72	Load/Store Instructions	FF		0x72	Load/Store Instructions	C7AE3

Table 12: Multiple Sobel and TEA With L2 Cache Disabled

Appendix C

1.1 Thesis Presentation

Technological Education Institute of Crete
School of Engineering
Department of Informatics Engineering

Design and Implementation Mechanisms for quality of service over Embedded systems on chips

Efstathia Matthaiou
AM 2930
Supervising Professor
George Kornaros

Overview

- Introduction
- Hardware Implementation Description
- Performance Monitoring Unit
- Applications/Benchmarks
- Software Implementation Description
- Measurements-Results
- Conclusions and Future Work

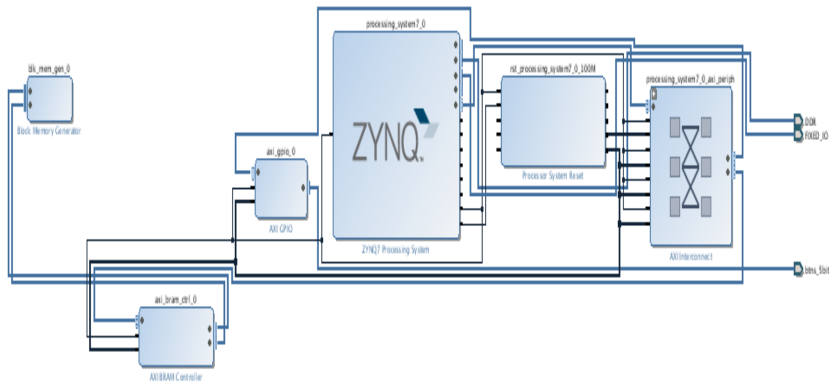
Aims and Objectives

- Implementation of applications on the hardware design
- Enablement of the Performance Monitoring Unit
- Measurements by the Performance Monitoring Unit
- Study on the results obtained

Hardware Implementation Description

- The implementation is carried out on the Zynq-7000 based on the Xilinx all programmable SoC (AP SoC) architecture
- Processing System(PS)
  - Dual-core ARM Cortex-A9 Processor
  - The Level-One (L1) Cache
  - The Level-Two (L2) Cache
  - The DDR Memory
  - ARM Private Timer
- Programmable Logic(PL)
  - AXI BRAM
  - AXI GPIO
  - Block Memory Generator

Hardware Design



Performance Monitoring Unit

- Hardware that has been built inside any given processor in order to measure its performance parameters
  - The Performance Monitor Unit (PMU) is part of the ARM Debug architecture

Performance Monitoring Unit Counters

- Six event counters (PMU event counter 0 to 5)
- Each one of the counters are able to count any of the 63 events available in the Cortex-a9 processor

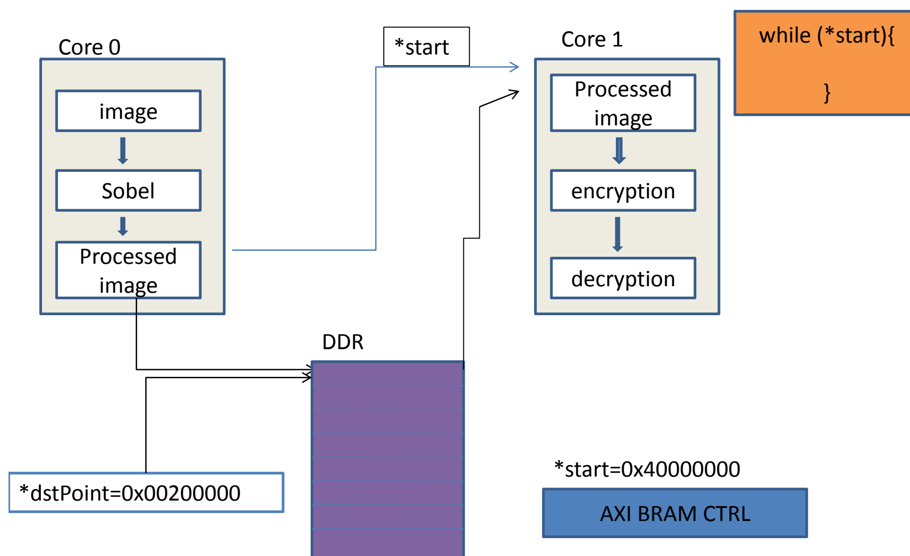
Performance Monitoring Unit Events

- Architectural Events
  - Instruction cache miss (0x01)
  - Data cache miss (0x03)
  - Data cache access (0x04)
  - Data read (0x06)
  - Data writes (0x07)
  - Cycle count (0x11)
- Specific Events
  - Load/Store instructions (0x72)
    - Counts the number of instructions being executed in the Load/Store unit
  - Processor stalled because of a write to memory (0x81)
    - Counts the number of cycles when the processor is stalled
  - Processor stalled because of instruction side main TLB miss (0x82)
    - Counts the number of stall cycles because of main TLB misses on requests issued by the instruction side
  - Processor stalled because of data side main TLB miss (0x83)
    - Counts the number of stall cycles because of main TLB misses on request issued by the data side

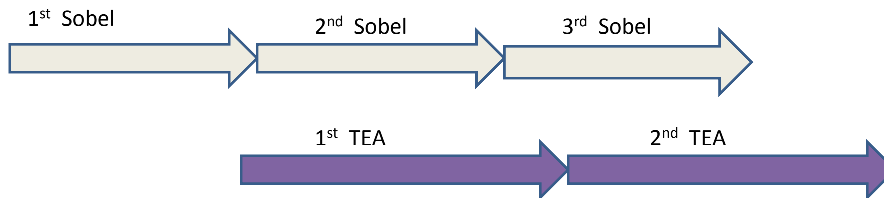
Application/Benchmarks

- Sobel
  - Image Processing
- TEA
  - Encryption
  - Decryption
- Black Scholes
  - European and call options formula
    - Asset Path
    - Binomial
    - BSF
    - Forward
    - MC

Single Sobel and TEA Application



Multiple Sobel and TEA Application



Performance Monitoring Unit Application

- Enablement
  - Enablement of the six event counter
  - Enablement of the cycle counter register
  - Reset of the event counter and ccnt to their original value
- Count
  - Configuration of the event counter for the specific events
  - Reading the event counters and ccnt before the main function
  - Reading the event counters and ccnt after the main function
- Disablement
  - Disablement of the cycle counter register
  - Disablement of the event counters

```
•void enable_pmu(void);
•void enable_ccnt(void);
•void enable_pmn(unsigned int counter);
```

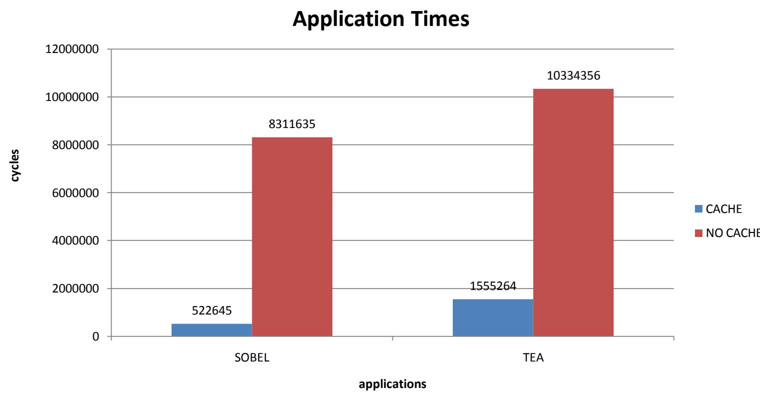
```
•void pmn_config(unsigned int counter, unsigned int event);
•unsigned int read_ccnt(void);
•unsigned int read_pmn(unsigned int counter);
```

```
•void disable_pmu(void);
•void disable_ccnt(void);
```

Formulas

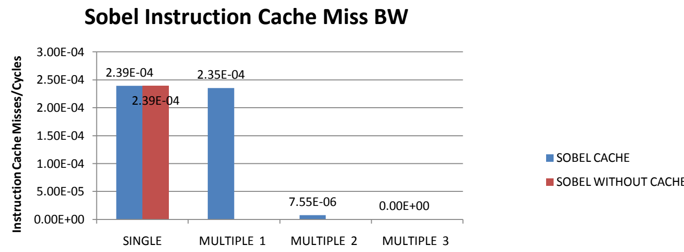
- The Cycle Counter Register (ccnt) provided by the PMU counts the same thing as the event Cycle Count (0x11). The equation linking these two is: cycle count/64=ccnt.
- The ScuTimer used is clocked at ½ of the CPUs frequency
- The Cortex-A9 CPU Clock Frequency is 666 Hz.
- The events Data read (0x06) and Data writes (0x07) added together equal the event Load/Store Instructions (0x72).
- The cache ratio is the Data cache misses (0x03) divided by the Data cache access (0x04).

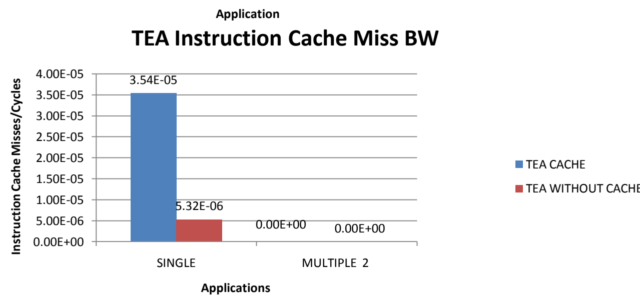
Sobel and TEA Application Time



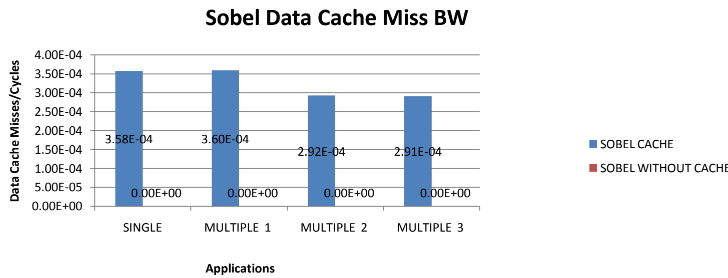
*The application times are the result of the “Cycle count” with a CPU Frequency at 666 Hz

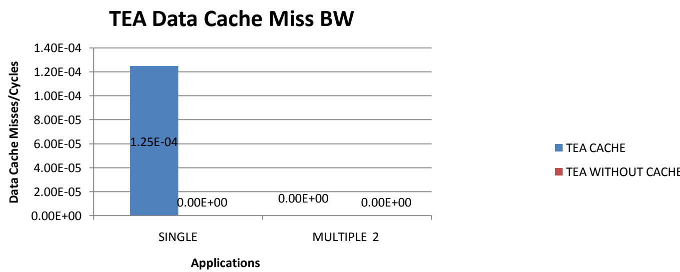
Sobel and TEA Instruction Cache Miss Bandwidth



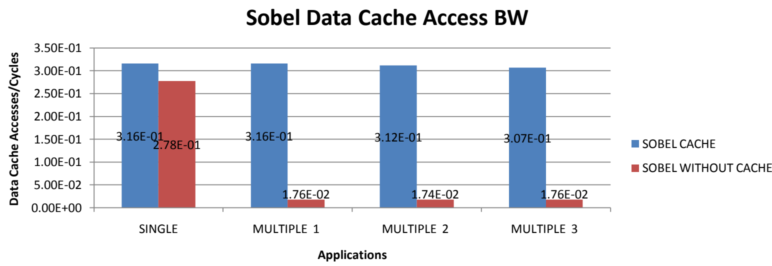


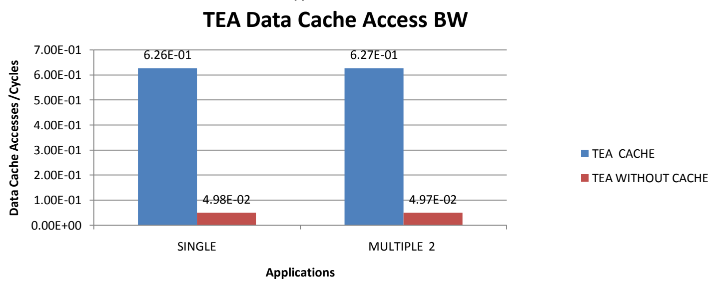
Sobel and TEA Data Cache Miss Bandwidth



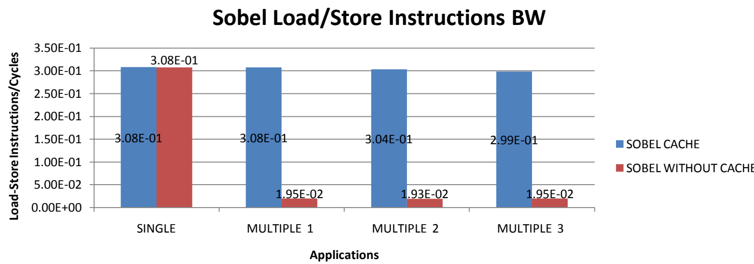


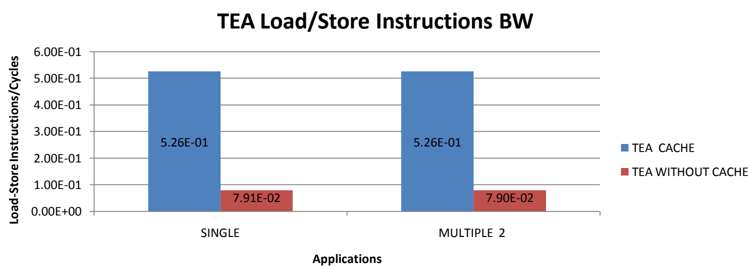
Sobel and TEA Data Cache Access Bandwidth



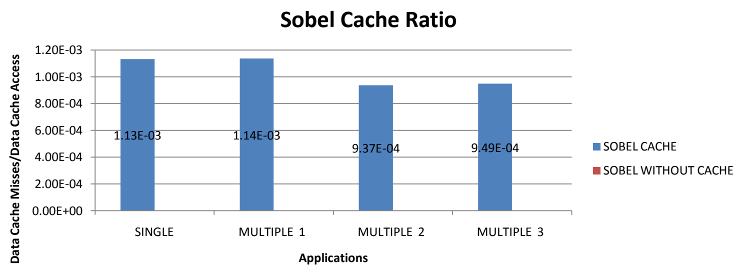


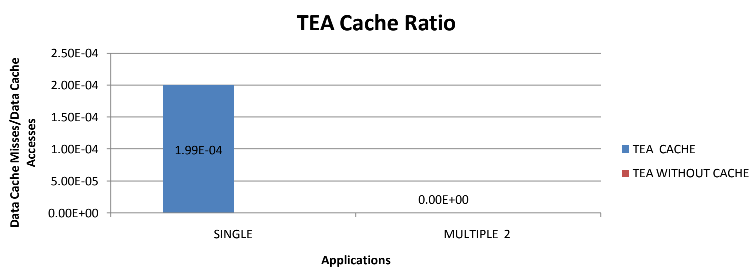
Sobel and TEA Load/Store Instructions Bandwidth



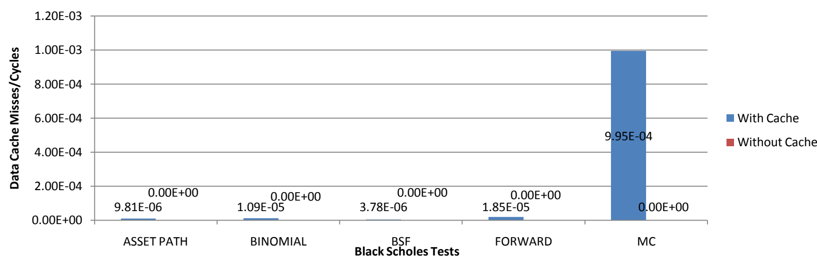


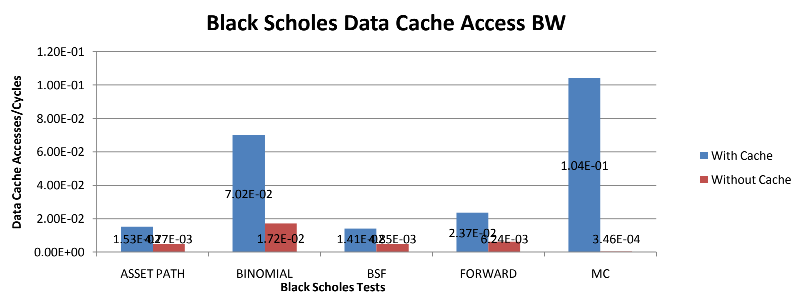
Sobel and TEA Cache Ratio



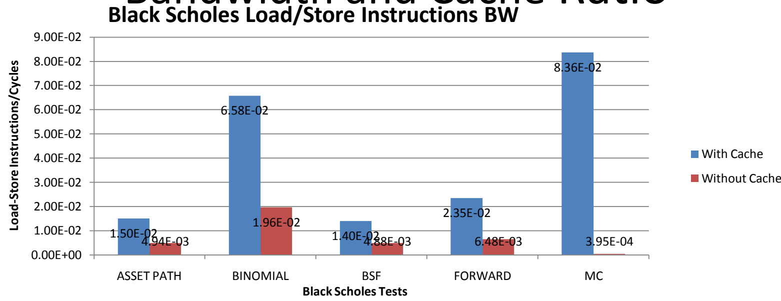


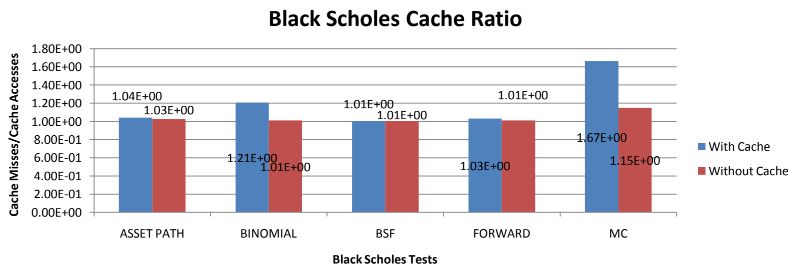
Black Scholes Data Cache Miss and Access Bandwidth



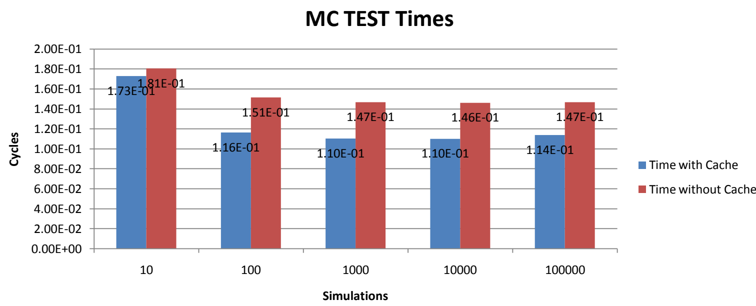


Black Scholes Load/Store Instructions Bandwidth and Cache Ratio





Black Scholes MC Application Time



Conclusions

- We are now able to understand and comprehend the consumption in our systems resources through the use of the Performance Monitoring Unit , therefore we could guide the processors to perform Quality of service.

Future Work

- We intend to focus on developing a system that will provide real-time quality of service. By using the Performance Monitoring Unit, the one core will run various applications and the other core will act as a monitor
- This can be expanded on to Linux Operating Systems, where this monitor will temporarily pause processes that have a high resource occupation in order to let other applications run

Thank you! ☺