CACHE OPTIMIZATION FOR REAL-TIME EMBEDDED SYSTEMS

by

Abu Asaduzzaman

A Dissertation Submitted to the Faculty of

College of Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

Florida Atlantic University

Boca Raton, FL

December 2009

CACHE OPTIMIZATION FOR REAL-TIME EMBEDDED SYSTEMS

by

Abu Asaduzzaman

This dissertation was prepared under the direction of the candidate's dissertation advisor, Dr. Imad Mahgoub, Department of Computer and Electrical Engineering and Computer Science, and has been approved by the members of his supervisory committee. It was submitted to the faculty of the College of Engineering and Computer Science and was accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

SUPERVISORY COMMITTEE:

Imad Mahgoub, Ph.D.
Dissertation Advisor

Borko Furht, Ph.D.

Ravi Shankar, Ph.D.

Mohammad Ilyas, Ph.D.

Valentine Aalo, Ph.D.

Borko Furht, Ph.D.
Chair, Department of Computer and Electrical Engineering and Computer Science

Karl K. Stevens, Ph.D., P.E.
Dean, College of Engineering and Computer Science

Barry T. Rosson, Ph.D.
Dean, Graduate College

Nov. 19, 2009
Date

VITA

Abu Asaduzzaman was born and raised in Faridpur, Bangladesh. He received the BS degree in Electrical and Electronic Engineering from Bangladesh University of Engineering and Technology (BUET), Dhaka in 1993 and the MS degree in Computer Engineering from Florida Atlantic University (FAU), Florida in 1997. Currently, he is working at FAU as Specialist Computer Applications. Previously (2003 – 2006), he worked for FAU Computer and Electrical Engineering and Computer Science Department and taught various hardware/software courses as Instructor. He, also, worked for BlueCross and BlueShield of Florida (2001 to 2003) and ECI Telecom (1998 to 2001) as IT Professional. His current research interests include modeling and simulation, cache memory optimization, multi-core architecture, and real-time embedded system. He has published several journal and conference papers in these areas. He has served as a session chair in various conferences including IMETI CCCT-2009 and IASTED PDCS-2008, both in Orlando, Florida. Mr. Asaduzzaman is a member of the IEEE, the honor society of Phi Kappa Phi, Tau Beta Pi, Upsilon Phi Epsilon, and Golden Key. He obtained the first place (in oral presentation) at FAU Graduate Research Symposium in 2005. He is one of the recipients of FAU Graduate Student Association's 2009 Owl Awards. Mr. Asaduzzaman's biography is published in the Who's Who Among Students in American Universities & Colleges 1997 and in the Who's Who in America 2010.

ACKNOWLEDGEMENTS

The author sincerely acknowledges the encouragement and guidance from his dissertation advisor Dr. Imad Mahgoub throughout this work. The author expresses his gratitude to the members of the supervisory committee for their guidance and valuable suggestions. The author is grateful to the Department of Computer and Electrical Engineering and Computer Science and the Office of Admissions for providing all kind of supports during this study and research work at Florida Atlantic University. The author thanks Motorola and Mirabilis Design for their supports to this work.

The author expresses his thanks to his wife Manira Rani and friends for their supports throughout the writing of this manuscript. Finally, the author acknowledges the destructions from some individuals throughout this work.

ABSTRACT

Author: Abu Asaduzzaman

Title: Cache Optimization for Real-Time Embedded Systems

Institution: Florida Atlantic University

Dissertation Advisor: Dr. Imad Mahgoub

Degree: Doctor of Philosophy

Year: 2009

Cache memory is used, in most single-core and multi-core processors, to improve performance by bridging the speed gap between the main memory and CPU. Even though cache increases performance, it poses some serious challenges for embedded systems running real-time applications. Cache introduces execution time unpredictability due to its adaptive and dynamic nature and cache consumes vast amount of power to be operated. Energy requirement and execution time predictability are crucial for the success of real-time embedded systems. Various cache optimization schemes have been proposed to address the performance, power consumption, and predictability issues. However, currently available solutions are not adequate for real-time embedded systems as they do not address the performance, power consumption, and execution time predictability issues at the same time. Moreover, existing solutions are not suitable for dealing with multi-core architecture issues.

In this dissertation, we develop a methodology through cache optimization for real-time embedded systems that can be used to analyze and improve execution time predictability and performance/power ratio at the same time. This methodology is effective for both single-core and multi-core systems. First, we develop a cache modeling and optimization technique for single-core systems to improve performance. Then, we develop a cache modeling and optimization technique for multi-core systems to improve performance/power ratio. We develop a cache locking scheme to improve execution time predictability for real-time systems. We introduce Miss Table (MT) based cache locking scheme with victim cache (VC) to improve predictability and performance/power ratio. MT holds information about memory blocks, which may cause more misses if not locked, to improve cache locking performance. VC temporarily stores the victim blocks from level-1 cache to improve cache hits. In addition, MT is used to improve cache replacement performance and VC is used to improve cache hits by supporting stream buffering. We also develop strategies to generate realistic workload by characterizing applications to simulate cache optimization and cache locking schemes. Popular MPEG4, H.264/AVC, FFT, MI, and DFT applications are used to run the simulation programs. Simulation results show that newly introduced Miss Table based cache locking scheme with victim cache significantly improves the predictability and performance/power ratio. In this work, a reduction of 33% in mean delay per task and a reduction of 41% in total power consumption are achieved by using MT and VCs while locking 25% of level-2 cache size in an 4-core system. It is also observed that execution time predictability can be improved by avoiding more than 50% cache misses while locking one-fourth of the cache size.

DEDICATION

This manuscript is dedicated to my daughter, late Huma Aisha Zaman (5 years), who was the driving force for me to start the Ph.D. program and my son, Rayan Mahdi Zaman, who is the driving force for me to finish it up!

CACHE OPTIMIZATION FOR REAL-TIME EMBEDDED SYSTEMS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Cache memory is first introduced by IBM in 1960s to improve performance by reducing the speed gap between the main memory and CPU. Almost immediately after that all gigantic chip-vendors introduced cache to their processors [59], [113], [158]. Today, processors are having multiple processing cores and most processors have on-chip level-1 cache (CL1) and off-chip level-2 cache (CL2) [35], [43], [102], [109], [110], [134], [166]. Some processors have even higher levels of caches – AMD Opteron has level-3 cache (CL3). Even though cache improves overall system performance, it makes the system more unpredictable and the system consumes more power [65], [74], [128], [131], [139], [144], [145], [163]. Excessive power consumption and execution time unpredictability may defeat the performance gain of embedded systems, especially when the system is battery-operated and runs real-time applications where predictability is crucial [17], [47], [50], [57], [70], [93], [127], [142], [147]. Special design considerations are required for successful implementation of cache memory subsystem for real-time embedded systems [56]. In this dissertation, we study currently available solutions and introduce new solutions to improve the predictability and performance/power ratio by optimizing the cache memory subsystem of real-time embedded systems.

1.1 Cache Memory in Computing Systems

IBM introduces the first cache memory (on-chip CL1) in the System/360 Model 85 in 1960s. In early 1990s, off-chip CL2 appears in Intel 486DX4 and Pentium chips. Digital Equipment Corporation presents Alpha 21164 with CL3 in 1995. Recently, various cache memory organizations are being used in both single-core and multi-core processors. Considering the cache organizations we summarize the computing systems as shown in Figures 1.1(a) for single-core and 1.1(c) for multi-core architectures. Usually each processor has on-chip CL1. Single-core processors may have higher levels of caches like CL2, CL3, etc. Multi-level caches in multi-core processors may be organized in a number of ways like shared CL2, dedicated CL2, etc. Cache architecture may be inclusive or exclusive and/or may have victim caches [see Figure 1.1(b)].

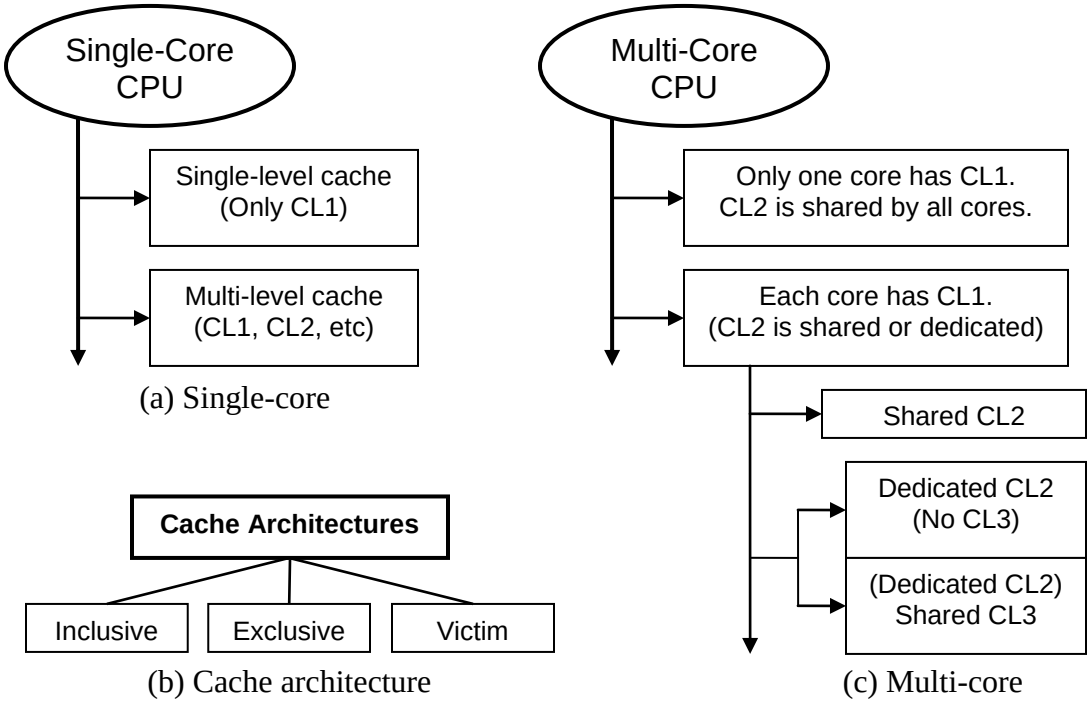


Figure 1.1: Contemporary processors and cache organizations.

1.1.1 Cache in Single-Core Architectures

A single-core processor with only on-chip CL1 has a simple architecture and may be good for small computing systems. However, most single-core processors have additional level(s) of cache(s) along with CL1. Multi-level cache architectures may be inclusive or exclusive and may have victim cache [134]. Intel Pentium 4, one of the most popular single-core processors of its time, uses inclusive cache architecture. The schematic diagram of typical inclusive cache architecture is shown in Figure 1.2. Here, CL2 contains each and every blocks that CL1 (i.e., I1 and D1) may contain. In case of a CL1 miss followed by a CL2 miss, the block is first brought into CL2 from main memory, then into CL1 from CL2. Intel Pentium 4 Willamette has on-die 256 KB inclusive level-2 cache; with 8 KB level-1 trace/instruction cache (I1) and 8 KB level-1 data cache (D1). The effective cache size of this architecture is CL2 size.

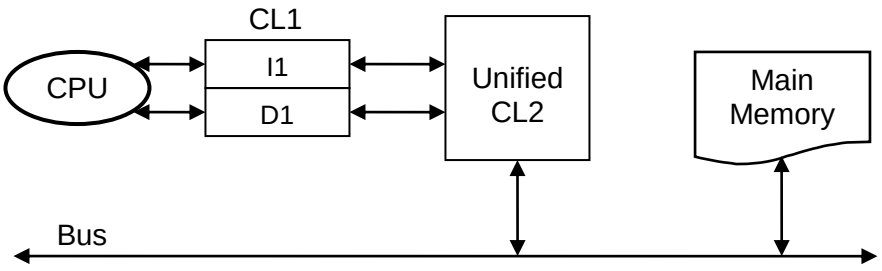


Figure 1.2: Inclusive cache architecture.

Figure 1.3 is the schematic diagram of a typical exclusive cache architecture where CL2 does not contain blocks that are in CL1 (i.e., I1 and D1) [155]. A victim buffer with CL1 may improve performance. In this architecture, in case of a CL1 miss followed by a CL2 miss, the block is directly brought into CL1 from main memory. As a result, the effective cache size is CL1 size + CL2 size for this architecture. AMD Athlon

uses exclusive cache architecture. AMD Athlon Thunderbird has on-die 256 KB exclusive level-2 cache with 64 KB I1 cache and 64 KB D1 cache.

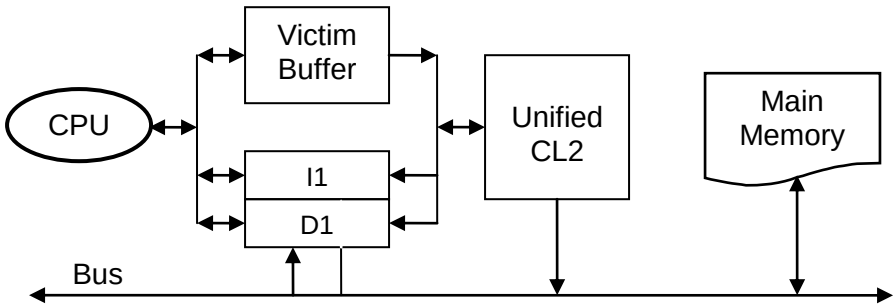


Figure 1.3: Exclusive cache architecture with victim buffer.

Figure 1.4 presents the schematic diagram of a cache memory system with a victim cache between CL1 (I1, D1) and CL2 [58]. Victim cache reduces average memory latency by temporarily holding the victim blocks from CL1. Usually the effective cache size of this architecture is more than CL2 and it provides performance gain. The exclusive and victim cache hierarchy is suitable for systems with limited cache-memory area (like embedded systems) and applications that perform a large amount of memory accesses (like multimedia) [155].

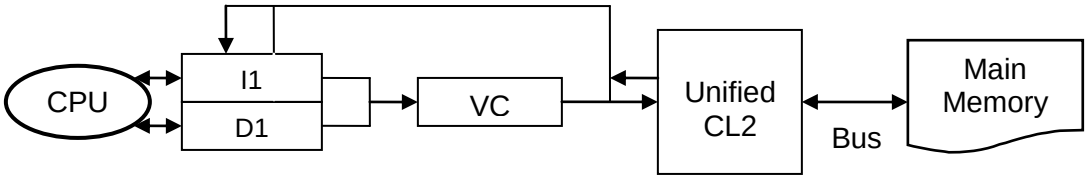


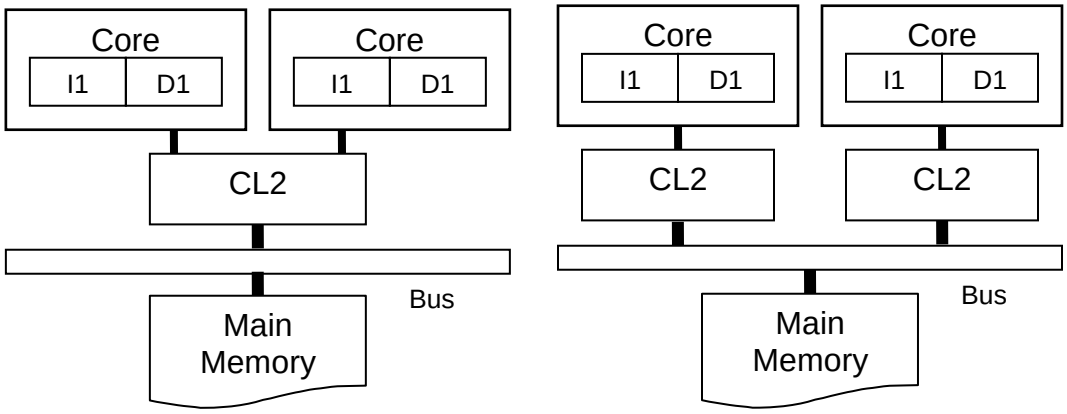
Figure 1.4: Cache architecture with victim cache.

1.1.2 Cache in Multi-Core Architectures

Most manufacturers are adopting multi-core processors to acquire additional processing speed for modern embedded systems. In a multi-core CPU or chip-level

multiprocessor (CMP), two or more independent cores are combined into a die [43], [100], [103], [104], [109], [132], [150], [166].

Figures 1.5(a) and 1.5(b) show the schematic diagram of Intel dual-core with shared CL2 (Dual-Core Xeon: 64 KB I1, 64 KB D1, 4 MB CL2) and AMD dual-core with dedicated CL2 (Athlon Classic: 64 KB I1, 64 KB D1, 512 KB CL2), respectively [166]. The processing cores share the same interconnect with the rest of the system.



(a) Shared CL2 (Intel - Xeon) (b) Dedicated CL2 (AMD - Athlon)

Figure 1.5: Dual-core architecture.

As shown in Figures 1.6 and 1.7, Intel quad-core (Xeon DP: 128 KB I1, 128 KB D1, 8 MB CL2) has shared CL2 [109]; but AMD quad-core (Opteron: 256 KB I1, 256 KB D1, 2 MB CL2) has dedicated CL2 and shared CL3 [110]. CL3 of AMD Opteron processors may be 2 MB (Santa Rosa) or 4 MB (Deerhound).

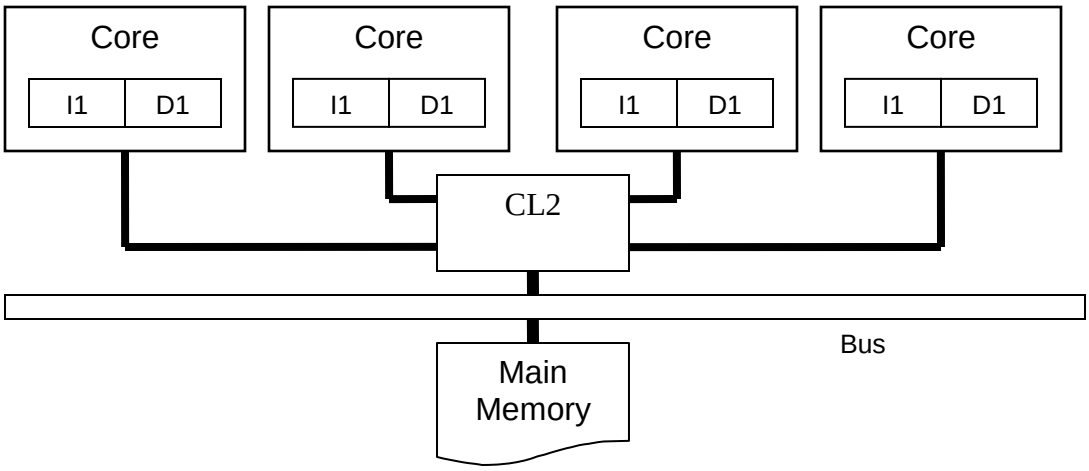


Figure 1.6: Quad-core architectures (Intel – Kentsfield XE).

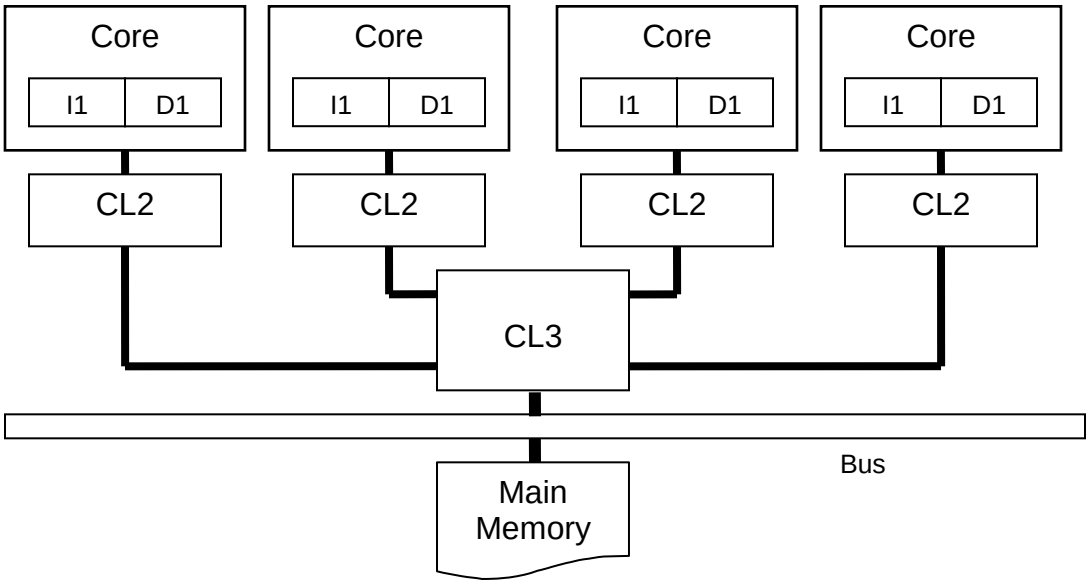


Figure 1.7: Quad-core architectures (AMD - Opteron).

Cell processor is developed by Sony, Toshiba, and IBM (STI) in a joint venture [19], [117], [124], [137]. Cell multi-core architecture primarily boosts up the processing speed demanded by the electronic games [3], [55]. Figure 1.8 shows a Cell-like architecture. Cell has a Primary Processing Entity (PPE) with dual cores and eight helper

units called Synergistic Processing Element (SPE). The PPE contains a 32 KB I1 and a 32 KB D1 caches. A 512 KB CL2 is shared by the PPE and SPEs. Primarily the PowerPC PPE keeps the processor compatible with lots of applications.

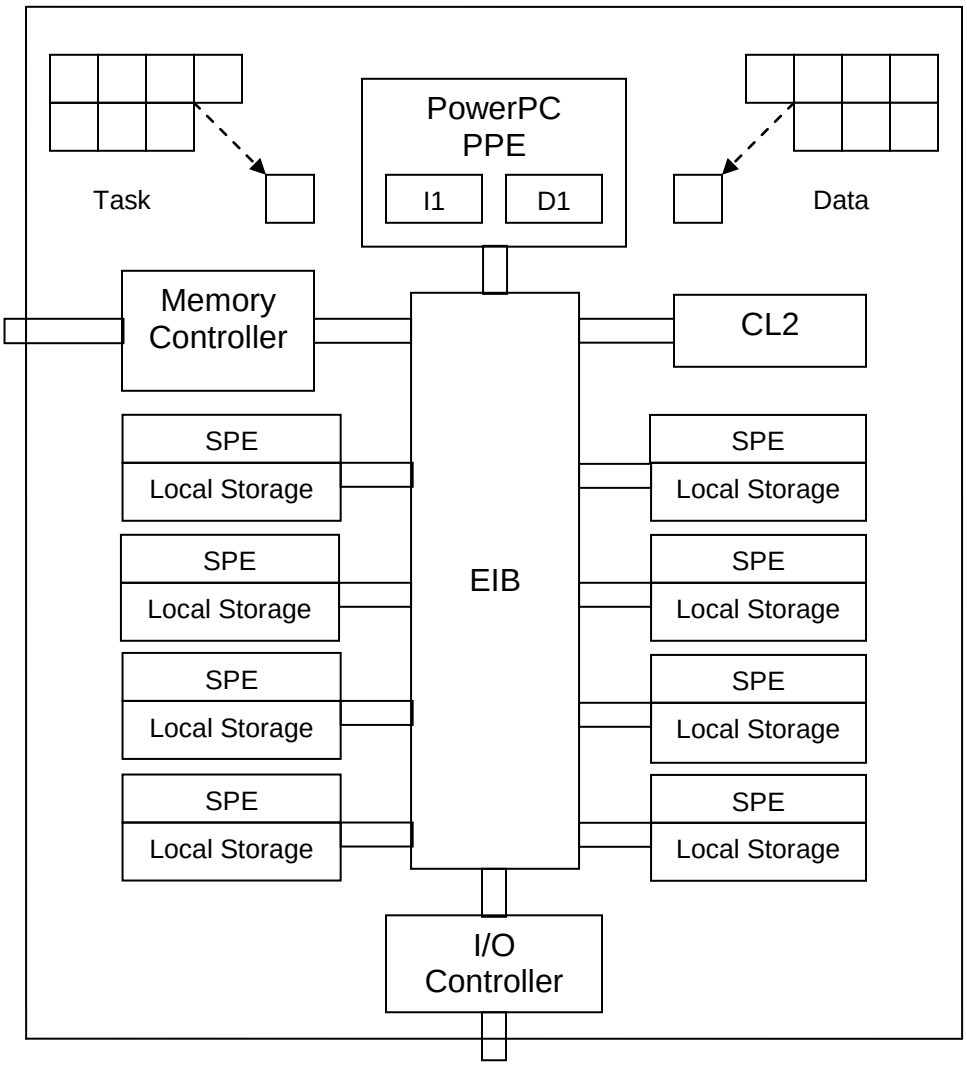


Figure 1.8: STI Cell-like architecture.

Each one of the 8 little helper cores (i.e., each SPE) is a "Cell". Each Cell has 256 KB SRAM and a 4 x 128 bit ALU (Arithmetic Logical Unit), and 128 of 128-bit registers. The Element Interconnect Bus (EIB) is the communication bus internal to the

Cell processor which connects various on-chip system elements like PPE processor, memory controller (MIC), eight SPE coprocessors, and two off-chip I/O interfaces.

Based on the above discussion, cache memory subsystem is an important component in both single-core and multi-core computing systems. A lot of research has been done to address the performance, power consumption, and predictability issues through cache optimization. We briefly summarize them in the following sections by mentioning how further improvement may be possible.

1.2 Cache Modeling to Enhance Performance

Cache modeling is helpful to analyze cache performance and suggest how to achieve significant execution speedup, particularly when the system is designed to run computation intensive applications. The computing speed of microprocessors has exponentially increased in the past few decades and so is the support to computation intensive embedded systems. With such improved computing power, memory subsystem deficiency (memory is slower than the CPU) becomes the major barrier to support real-time multimedia applications on embedded systems [5], [6]. Studies show that for most applications there are sufficient reuses of values for caching [7], [82], [119], [121]. Hence, there is an opportunity for customizing the cache memory subsystem through cache modeling for improved performance.

1.3 Cache Optimization to Improve Performance/Power Ratio

Traditionally there are tradeoffs between the performance and power consumption [69]. Cache memory optimization techniques are becoming popular due to their guiding

ability to increase the system performance and decrease total power consumption [148]. To satisfy the needs for increased processing power, embedded systems are using multi-core (instead of single-core) processors. The reason behind this trend is twofold – first, the popularity and demands of embedded systems are increasing every day; second, billions of transistors are possible in a single chip. This trend is expected to increase for the next decade. In multi-core architecture, performance is improved because the application is divided into smaller tasks and tasks are assigned among multiple cores. Total power consumption is reduced because the system runs at a lower frequency [20], [60], [96], [126]. However, multi-level caches in multi-core systems consume huge amount of power [7]. Therefore, there are opportunities to increase performance and decrease total power consumption by cache optimization in multi-core systems.

1.4 Cache Locking to Enhance Predictability

Even though cache improves performance (by reducing the speed gap between the main memory and CPU), the execution time becomes unpredictable due to cache's adaptive and dynamic behavior [11]. Real-time embedded systems work under real-time constraint – the hardware and software are subject to operational deadlines from event to system response. Execution time predictability is a crucial factor for the success of these complex systems [12], [21]-[29], [38], [99], [130]. Studies show that cache locking helps reduce the worst case execution time (WCET) and cache-related preemption delay [52], [97], [105], [128]. Therefore, cache locking mechanism (along with other techniques like selective pre-loading) can be used to improve execution time predictability in real-time embedded systems.

1.5 Problem Statement

An embedded system is a special-purpose computing system embedded in an electrical or mechanical device to perform one or a few dedicated tasks repeatedly. Traditionally, embedded systems suffer from limited resources like power supply and memory bandwidth [15], [62], [80], [81], [141], [160], [167]. However, embedded systems are becoming more popular and more useful in various vicinity of life including medical treatments, entertainment, and other scientific activities. Modern embedded systems are made to support various complicated real-time applications where the completion of an operation after its deadline is considered useless and that may lead to a critical failure of the complete system [76], [101], [116], [170]. Therefore, performance, power consumption, and predictability – all are important factors for designing future real-time embedded systems.

Cache memory is introduced to improve performance by reducing the speed gap between the main memory and CPU. Even though embedded systems take advantage of using caches in their architectures (single-core or multi-core), cache poses some serious challenges for embedded systems running real-time applications. Due to cache's adaptive and dynamic behavior, cache increases execution time unpredictability. Also, cache is power-hungry and consumes substantial amount of power to be functional. If not properly designed, excessive power consumption and execution time unpredictability may defeat the performance gain by cache in real-time embedded systems.

Cache parameters have influence on the performance, power consumption, and predictability of any computing system. It has been established that system performance can be increased and power consumption can be decreased by customizing the cache

parameters, using victim cache, and applying stream buffering [5], [6], [10], [32], [33], [35], [44], [49], [51], [56], [58], [74], [82], [113], [122], [155], [158], [163]. Studies also indicate that cache-locking along with selective pre-loading improves predictability when the right cache blocks are selected [9], [11], [12], [21], [24], [26], [28], [97], [128], [139]. However, there are tradeoffs between the predictability and performance/power ratio. The influence due to the presence of cache is very significant for real-time embedded systems where performance, power consumption, and predictability – all are important. Currently available solutions are not capable of analyzing the performance, power consumption, and predictability at the same time. Moreover, existing solutions do not address multi-core architecture issues very well.

Therefore, even though cache improves performance, there are serious problems for real-time embedded systems due to the presence of cache as cache makes execution time unpredictability worse and consumes large amount of power. There is no efficient solution to determine the optimal cache parameters to achieve the best predictability and performance/power ratio at the same time for both single-core and multi-core embedded systems running real-time applications.

1.6 Major Contributions of this Dissertation

In this dissertation, we develop a cache modeling and optimization methodology using Miss Table based cache locking and other techniques like stream buffering to improve the predictability and performance/power ratio for real-time embedded systems. Major contributions of this research work are listed below:

1) We develop cache modeling and optimization technique to enhance overall system performance of single-core systems. With improved CPU speed, memory sub-system deficiency is the major barrier to improving the system performance. Studies show that there is sufficient reuse of values for caching that may significantly reduce the memory bandwidth. This cache modeling technique helps enhance performance by efficiently selecting the cache parameters for the target applications.

2) We develop a methodology using cache modeling and optimization technique that improves performance/power ratio of multi-core embedded systems. To satisfy the need for increased processing power, embedded systems are adopting multi-core (instead of single-core) processors. Like single-core systems, the presence of cache in multi-core embedded systems poses a number of challenges including power consumption. This cache optimization technique effectively improves performance/power ratio for multi-core embedded systems.

3) We develop a novel cache locking scheme that can be used to improve execution time predictability of a real-time embedded system. Even though cache memory improves performance by reducing the speed gap between the main memory and CPU, the execution time becomes unpredictable due to the cache's adaptive and dynamic behavior. Execution time predictability is a crucial factor for the success of real-time systems. This cache locking scheme helps improve predictability.

4) We propose a methodology that is effective to improve the predictability and performance/power ratio for both single-core and multi-core systems by introducing Miss Table at cache level to help cache locking and by using victim cache(s) to temporarily store the victim blocks. In addition, Miss Table helps improve cache replacement

performance and victim caches help improve performance/power ratio by supporting stream buffering. This Miss Table based cache locking scheme with victim cache(s) has potential to improve the predictability and performance/power ratio for both single-core and multi-core real-time embedded systems.

5) We introduce various techniques to characterize applications and generate realistic workload to evaluate proposed cache optimization solutions. For cache locking, we develop a 3-phase workload characterization strategy. In phase-I, we divide big applications into smaller end-to-end functions. In phase-II, we estimate major operations (like integer, floating-point, load/store, branch, etc) in the code segment. Finally in phase-III, we select the blocks (to be locked) that may create more misses if not locked.

1.7 Organization of this Dissertation

This dissertation presents a cache modeling and optimization methodology using Miss Table based cache locking with victim caches to improve the predictability and performance/power ratio for real-time embedded systems. The following is an outline of the dissertation:

- Chapter 1 is the introduction. It introduces the topic and motivates the need for cache optimization in real-time embedded systems including the problem statement. It states the goal and presents the contributions of this work.
- Chapter 2 presents a comprehensive survey of related articles.
- Chapter 3 presents a cache modeling and optimization technique that enhances performance of a single-core system. Basic cache memory subsystem, simulation details, and simulation results are discussed to show how this

single-core cache modeling and optimization technique can be useful to determine the optimal cache parameters for target applications.

- Chapter 4 discusses a cache modeling and optimization technique to improve the performance/power ratio of a multi-core system. Basic multi-core cache memory organization, modeling multi-core architecture, and experimental results are discussed. Experimental results show that this cache modeling and optimization strategy is useful to analyze the performance and power consumption of multi-core systems running real-time applications.
- Chapter 5 implements a cache locking technique that increases the predictability by locking important blocks. Basic cache locking information, simulation details, and experimental results are discussed to show how this cache locking scheme can be used to improve execution time predictability of a single-core real-time embedded system.
- Chapter 6 introduces a Miss Table based cache locking with victim cache, suitable for both single-core and multi-core systems, to improve the predictability and performance/power ratio. For a multi-core architecture, cache locking using MT, stream buffering using VCs, workload characterization for cache locking, simulation details, and experimental results are discussed. Simulation results show that MT and VCs help improve the predictability and performance/power ratio of embedded systems running real-time applications.
- Chapter 7 concludes this dissertation and discusses possible future extensions.

CHAPTER 2

LITERATURE SURVEY

In this work, our goal is to optimize cache memory organization of real-time embedded systems to improve execution time predictability and performance/power ratio. As cache is proven to be an important component to improve the overall system performance, it has been attracting researchers since it was first introduced by IBM in 1960s. We perform an extensive literature survey to see the progress and outstanding issues regarding cache organization and real-time embedded systems. A lot of research has been done and significant progress has been made to address the performance, power consumption, and predictability issues in embedded systems. However, most of the work that has been done focuses on single-core architectures and not suitable for multi-core architectures. Some work focus on only performance issues, some other work focuses on only power issues or only predictability issues. Also, various cache memory hierarchies have been proposed for various purposes. Different evaluation techniques, simulation tools, and workload have been developed [39], [51]. To summarize our literature survey, we classify them in 6 major categories – cache memory, embedded system, performance-power-predictability, workload, simulation, and other related issues. In Figure 2.1, we present these categories and their sub-categories (if any).

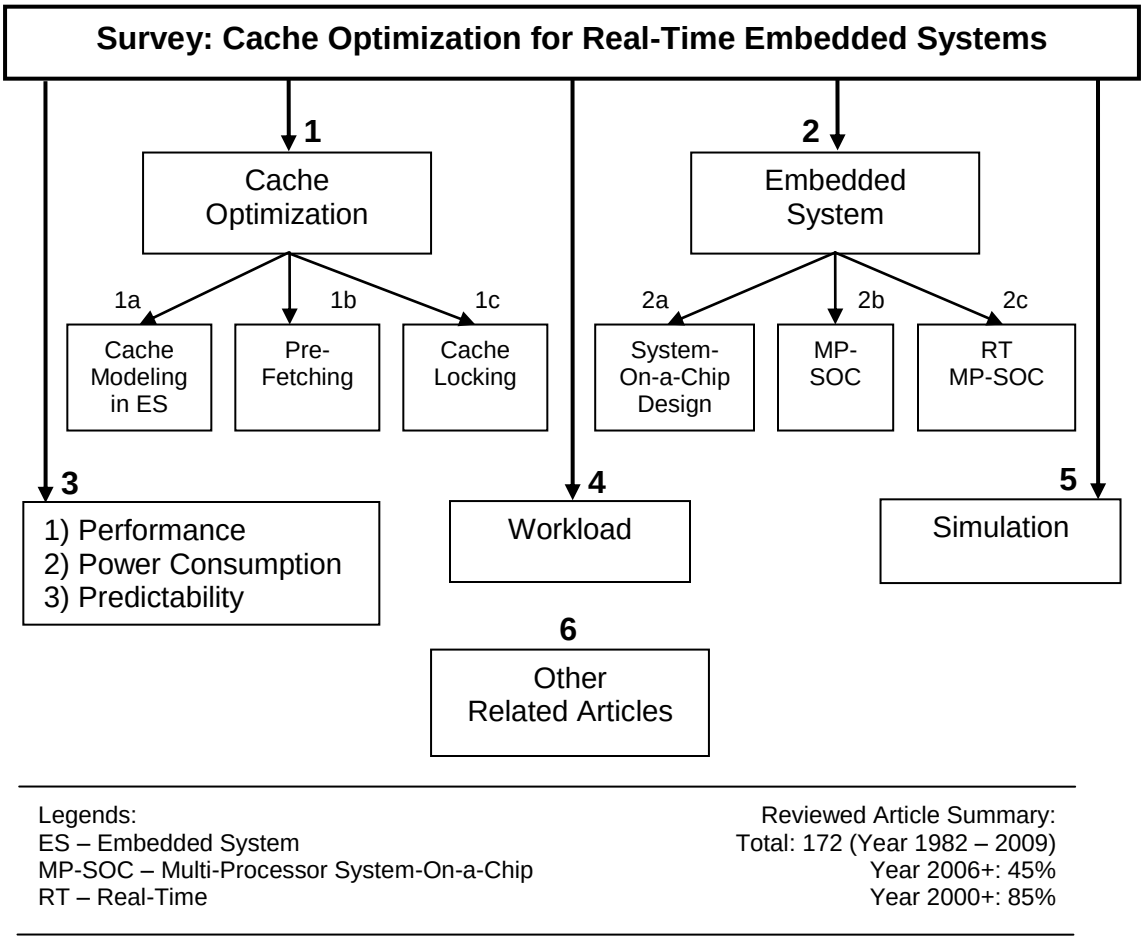


Figure 2.1: Summary of articles surveyed.

We study over 172 articles published in various journals and conference proceedings including IEEE and ACM publications, online postings, and whitepapers from big corporations including Intel, AMD, IBM, and Motorola. Over 45% articles are very recent (published in year 2006 or later) and over 85% articles are publishes in year 2000 or later. We select a few of these articles, very related to our work, to discuss in this chapter. We include 172 references in the Bibliography at the end of this manuscript.

2.1 Cache Optimization

Cache memory has a very rich history in the evolution of modern computing. In this subsection, we focus on cache parameters, cache organizations, victim cache, pre-fetch, cache locking, and cache optimization techniques. Cache memory is first seen in the IBM System/360 Model 85 in 1960s. In 1989, Intel 468DX microprocessor introduced on-chip 8 KB CL1 cache for the first time. In early 1990s, off-chip CL2 cache appeared with 486DX4 and Pentium microprocessor chips. Today's microprocessors usually have 128 KB or more of CL1, and 512 KB or more of CL2, and optional 2 MB or more CL3 [30], [33], [158]. Some CL1 cache is split into I1 and D1 in order to improve performance [110]. Important cache parameters include cache size, line size (aka, cache block size), and associativity level. Figure 2.2 illustrates various cache parameters. For a cache with S sets and W levels (or degrees) of associativity, total number of blocks (aka, lines), B = S x W and total cache size, C = B x Sb. Here, Sb is the size of a block (i.e., line size). For direct mapping, W = 1 and S = B; for set-associative, 1 < W < B; and for fully-associative, W = B and S = 1. Studies indicate that the predictability and performance/power ratio can be enhanced by optimizing the cache parameters.

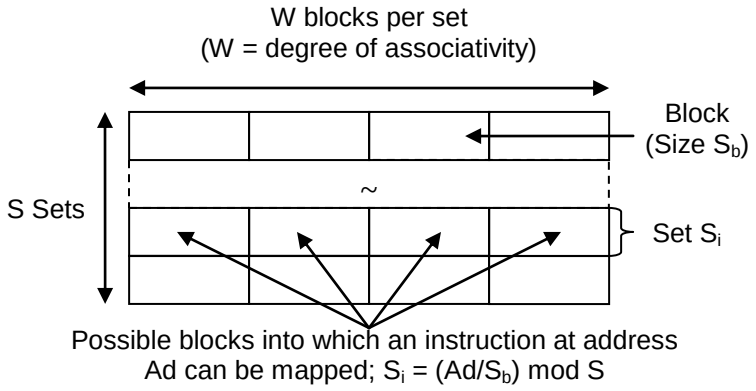


Figure 2.2: Cache parameters – cache size, line size, and associative level.

A multi-level cache memory organization can be inclusive or exclusive. In an inclusive cache architecture with CL1 and CL2, CL2 contains each and every block that CL1 (i.e., I1 and D1) may contain. In [134], Intel Pentium 4 (IP4) processor is discussed. IP4 is a popular single-core processor that uses inclusive cache architecture. In case of a CL1 miss followed by a CL2 miss, the block is first brought into CL2 from main memory, then into CL1 from CL2. IP4 Willamette has on-die 256 KB inclusive level-2 cache; with 8 KB level-1 trace/instruction cache (I1) and 8 KB level-1 data cache (D1). The effective cache size of this architecture is CL2 size. Unlike inclusive cache architecture, CL2 does not contain blocks that are in CL1 (i.e., I1 and D1) in exclusive architecture. In [155], performance evaluation of exclusive cache hierarchies is conducted. Here, in case of a CL1 miss followed by a CL2 miss, the block is directly brought into CL1 from main memory. As a result, the effective cache size is increased to CL1 size + CL2 size. AMD Athlon Thunderbird architecture has on-die 256 KB exclusive level-2 cache (with 64 KB I1 + 64 KB D1). In [4], victim buffer with exclusive cache offers reduction in power consumption with comparable performance gain. In [153], experimental results show that on PowerStone and MediaBench benchmarks, victim buffer can reduce energy consumption by 43%. However, the increased complexity of the exclusive cache hierarchy needs to be justified against the system and applications. Sometimes scratchpad (memory) is used to improve performance [1]. However, scratchpad memory may degrade performance when loading large information due to fragmentation.

In case of a CL1 miss, a new block should be brought into CL1. If CL1 is full, a block from CL1 is selected (depending on the cache replacement policy like random,

LRU – Least Recently Used, or FIFO – First In First Out) to be replaced by the new block (from CL2). The selected block is called victim block. A victim cache is introduced between CL1 (I1, D1) and CL2 in [58] to store the victim blocks for future usages. Victim cache reduces average memory latency. Usually the effective cache size of this architecture is more than CL2 size and it provides performance gain. Cache size cannot be increased without limitation. A larger cache usually costs more and becomes slower. The victim cache hierarchy is suitable for systems with limited cache-memory area (like embedded systems) and applications that perform a large amount of memory accesses (like multimedia) [155]. Performance improvement due to victim cache with smaller cache sizes and/or running applications with good caching behavior is most significant. Victim cache hierarchy significantly reduces execution time and improves predictability. In [4], victim cache showed the best performance among other cache systems like Half-and-Half and Cooperative for applications that require high memory bandwidth and low memory latency to obtain good performance.

Pre-fetching is a technique that allows memory subsystem to import data into the cache before the processor needs it [36], [72]. The selective pre-fetching proposed by [94] uses a history of references to decide on what block to be pre-fetched on a miss. Results show that pre-fetching may improve predictability for hard real-time systems. Aggressive pre-fetching can lead to cache pollution and also increase memory traffic. In distributed shared memory (DSM) systems, remote memory accesses take much longer than local ones, and hence data pre-fetching should be effective for such systems [46]. In [64], data pre-fetching issues on DSM systems are studied. They develop a new memory consistency semantic (MCS) model under which the pre-fetchable shared data objects, as

well as the best moment to launch a pre-fetching operation, can be easily identified. Simulation results show that this pre-fetching approach, combined with aggressive consistency, can substantially improve the performance of DSM systems. The pre-fetching scheme in [45] maintains a weighted directed flow graph whose vertices represent the objects accessed by the users and whose edges represent the reference flow relationships between these objects. Each edge is weighted in proportion to the frequency of reference. In [87], a global history buffer (GHB) is used to organize data cache pre-fetch information. GHB helps real-time system design by predicting cache misses accurately. Adaptive pre-fetching technique is investigated in [95]. They use pre-fetch counter and pre-fetch coefficient counter to make the cache controller adaptive. Results show significant improvement for instruction caches.

It is extremely important for (hard) real-time systems to make the memory access time predictable. In order to improve execution time predictability in multitasking hard real-time systems, [99] proposes that their content can be statically locked such that memory access time and cache-related preemption times are predictable. Cache locking is a technique to hold some or all cache blocks for the entire execution time. In [128], a memory hierarchy is proposed to build such that it provides high performance combined with high predictability that can be used complex system analysis. As mentioned in [129], the system performance depends on the instructions loaded and locked in cache for locking caches. In [130], an invalidation lock mechanism is implemented that utilizes the exclusive state of the snooping cache. Lock operations are frequent on the Parallel Inference Machine (PIM). Experimental results demonstrate the benefits of the lock mechanism for a few lock contentions and confirm that, in most cases, the lock

mechanism works well on the PIM. However, the mechanism may cause performance degradation in a tightly-coupled multi-processor (TCMP) in case of heavy contention. Interests in cache locking in multi-core systems are growing among the researchers. A brief introduction is given in [34] to the different methods of locking in Windows and the associated performance costs applicable to future multi-core architectures. An alternative approach is presented in [97] to cope with the predictability problem due to caches in real-time systems to statically lock their contents so as to make memory access times and cache-related preemption times entirely predictable. More study is needed to see the impact of this approach on performance for larger real benchmarks and the applicability of static cache locking techniques to data/unified/multi-level caches. An algorithm is proposed in [98] for off-line selection of the contents of two on-chip memories - locked caches and scratchpad memories. Experimental results show that the algorithm yields to good ratio of on-chip memory accesses on the worst-case execution path, with a tolerable reload overhead, for both types of on-chip memories. Furthermore, worst-case performance with scratchpad memories may degrade when loading large information due to fragmentation. Also, worst-case performance with locked caches may degrade with large cache lines due to cache pollution. Static cache analysis with data cache locking is combined in [139] to estimate the worst- case memory performance (WCMP) in a safe, tight, and fast way. Experimental results show that this scheme is more predictable, without compromising the performance of the transformed program. Various cache locking algorithms are studied in [21] through [29]. Their contributions include a methodology to select a set of instructions to be preloaded in the cache using a genetic algorithm. This algorithm is to select a set of instructions to be locked in cache for better

performance and simultaneously estimate a tight upper bound of the response time of tasks. Experimental results show that in a variety of scenarios, this method obtains better performance than non-locking caches. An algorithm is proposed in [12] that partitions the task into a set of regions. Each region owns statically a locked cache contents determined offline. Performance improvement is observed, as compared with a system without any cache. Contrary to cache analysis techniques, this algorithm depends neither on the scheduling policy, nor on the cache line replacement policy. A new locking protocol is proposed in [52], called waiting processor variable (WPV) lock mechanism, which has only one lock-read bus traffic command. The WPV mechanism also uses the cache state lock mechanism to reduce the locking overhead and guarantees the FIFO lock operations in the multiple lock contentions. The simulation results on the WPV lock mechanism show that about 50% of access time is reduced comparing with the conventional queuing lock mechanism.

According to [89], in order to justify a cache memory architecture two assessments are needed: an evaluation of the performance gain for a specific application and an indication of the modifications required in the software. For example, IBM Cell architecture is expected to improve the performance of some applications but current application software is required to be rewritten to take advantage of multi-core architecture. Various contemporary techniques for optimizing data and memory used in embedded systems, including platform-independent and platform-specific, are discussed in [90]. Using trace-based techniques and WCET analysis performance can be measured. Experimental results show that cache not only improves performance, but reduces energy consumption. A general-purpose computing platform running MPEG2 application is

studied in [121]. The system includes a single-core CPU with unified CL1 and CL2. Experimental results show that there is sufficient reuse of values for caching to significantly reduce the raw required memory bandwidth for video data. The addition of a larger second level cache to a small first level cache can reduce the memory bandwidth significantly. Cache memories for embedded applications can be designed to increase performance while reduce area and energy consumed. They suggest that techniques like cache locking, selective caching, pre-fetching, cache partitioning, and data reordering may improve cache performance for application specific systems. In [119], cache behavior of multimedia applications is examined. This study indicates larger data cache line sizes than are currently used would be beneficial in case of multimedia applications. Cache memories for embedded applications can be designed to increase performance while reducing area and energy consumed [85]. It is shown that separating data cache into an array cache and a scalar cache can lead to significant performance improvements for scientific benchmarks. Such a split data cache can also benefit embedded applications. Other techniques to improve performance include ‘cache partitioning technique’, ‘column caching mechanism’, ‘array cache’, and so on. In [82], A.M. Molnos et al use cache partitioning techniques to find a static task execution order for inter-task data cache misses. Due to the lack of freedom in reordering task execution, this method optimizes the caches more. Proposed ‘column caching mechanism’, in [32], enables cache partitioning so that data with different locality can be isolated for improved performance. Also, columns can emulate scratchpad memory which is used extensively to improve predictability in embedded systems. During the execution of a program, the data stored in

columns can be explicitly managed as in a scratchpad or can be implicitly managed as in a cache and the management can be dynamic at small intervals.

Memory pressure for chip-multiprocessors can be reduced by using cache injection [168]. This is a complicated procedure and the performance benefit is not significant. The performance improvement depends on the right combination of pre-fetch and injection mechanism.

In summary, cache is an essential element in all modern computing systems and it has significant impact on the performance, power consumption, and predictability. After knowing the pros and cons of various existing cache optimization techniques, we introduce an efficient cache optimization and cache locking technique that can be used in multi-core systems, as well as single-core systems, to improve the predictability and performance/power ratio of embedded systems running real-time systems.

2.2 Embedded System

Embedded system is a special-purpose computing system embedded in an electrical or mechanical device to perform one or a few dedicated tasks, sometimes with real-time computing constraints. As mentioned in [160], computers in their earliest years in the 1940s, were sometimes dedicated to specific tasks, but were too large to be considered "embedded". The first embedded system is, probably, the Autonetics D-17 guidance computer released in early 1960s. Today embedded systems are everywhere including consumer electronics, home appliances, automobiles, and medical equipments.

Various embedded system design issues are discussed in [62]. Unlike desktop computing, building the fastest CPU and supporting it for the maximum computing speed

are not the primary concerns for designing embedded systems. In embedded systems, the combination of the external interfaces and the control algorithms is very important. The CPU is needed to implement various functions. Power management is extremely important in embedded systems to conserve battery power and to minimize heat production. In order to design profitable embedded systems, designers often review the past deployment to take various lifecycle issues into account.

System-on-a-chip (SOC) integrates all components of a computing system into a single integrated circuit [73], [84], [169]. SOC may contain one or more processing core(s) along with other required functions on the same chip. SOC architecture is the underlying architecture for many embedded systems and scalable supercomputers. Due to the success of the semiconductor industry, SOC is expected to grow into general purpose computing as well. Xpipes is a network-on-chip (NOC) architecture for gigascale systems-on-chip (SOC) [16]. Various design methodologies of NOC architecture are presented in [63], [133], [138], [151], [154].

Even though single-chip implementation may be beneficial for some portable systems to meet low-cost and low-power requirements, multi-core is the future of computing systems in all environments. Most researches are focusing on concurrent hardware and software design early in the development process. A global solution is suggested in [57] – multi-processor SOC (MPSOC) with up to eight computing nodes and a flexible interconnection network.

The move toward chip-level multiprocessing architectures with a large number of cores continues to offer dramatically increased performance and power characteristics. Some significant challenges presented by this move are discussed in [47].

A comparative energy and performance analysis of cache-coherence support schemes in MPSOCs is provided in [70], [125]. They implement hardware-based, software-based, and operating system (OS) based approaches. The OS-based solution is very expensive. Hardware-based solution needs more power when traffic grows. Software-based solution is feasible even when the architecture becomes complex.

Interconnection speeds are not scaling well with the increase of the number of cores on chip multi-processors (CMP). As a result, coherence is becoming a central issue for multi-core performance. A proximity-aware directory-based coherence scheme is evaluated in [20] to improve the performance of parallel programs on such processors. The proposed schemes result in speedups up to 74.9% for the workload used.

A hardware/software methodology is proposed in [126] to make caches coherent in heterogeneous multiprocessor platforms with shared memory. Up to 58% performance improvement is achieved with low miss penalty at the expense of adding simple hardware, compared to a pure software solution. Speedup can be improved even further as the miss penalty increases. This approach provides embedded system programmers a transparent view of shared data, removing the burden of software synchronization.

Future Dense-CMP (D-CMP) systems with 16 or more processor cores impose new design restrictions. Performance evaluation of two proposed D-CMP architectures is presented in [141] - the Shared Bus Fabric (SBF) architecture features a snoop cache-coherence protocol and is based on a high-performance bus fabric interconnection network. The second architecture follows a directory-based approach and integrates a bi-dimensional mesh as the interconnection network. Results show that the directory-based architecture outperforms the SBF one.

A design flow is presented to achieve a very short design cycle for application-specific multiprocessor architecture [71]. The design flow extracts the architectural parameters and instantiate architectural components including processors, memory hierarchy, and network topology. The flow generates communication coprocessor that adapts the processors in an application-specific way.

A Two-Level Memory Management hierarchy is proposed in [115] to deal with global on-chip memory allocation/de-allocation in a dynamic yet deterministic way in the billion transistors MPSOC designs. In this way, heterogeneous processors in a SOC can request and be granted portions of the global memory in twenty clock cycles in the worst case for a four-processor SOC, which is at least an order of magnitude faster than software-based memory management.

According to [123], it is possible to control the cache activities, and thereby improve the predictability of the real-time system by using a real-time kernel in the hardware system.

Modern embedded systems are adopting multi-core architectures to satisfy the need for improved performance/power ratio. However, (multi-level) caches (in multi-core architecture) make the unpredictability even worse. Proposed cache optimization and cache locking techniques should improve the predictability and performance/power ratio of both single-core and multi-core real-time embedded systems.

2.3 Performance-Power-Predictability

Performance evaluation is one of the primitive research areas in computing history. Power consumption is crucial for embedded systems as they suffer from limited

resources. Execution time predictability is drawing a lot of attention because of its importance in systems supporting real-time applications.

The design of cache memory hierarchy is a critical issue in embedded systems [122]. In [127], trace-based statistics and flexible visualization techniques are suggested to analyze multi-core systems running multimedia applications on the same chip. Due to the increase in complexity, the latest embedded system design requires a higher-level of abstraction. If the design starts at a higher level of abstraction, the process towards the selection of the optimal target architecture, as well as the partitioning of the functionalities, can be considerably accelerated [14]. By knowing the cache miss-ratio, performance can be estimated in advance and can be used as input for compilers and system developers. A static method that bounds the worst-case instruction cache miss-ratio of a program is presented in [114]. This method first constructs a control flow graph (CFG) of the program at machine code level. Finally it constructs a binary tree of possible execution paths that might lead to a worst case cache miss-ratio.

In general, there is a tradeoff between the cache performance and the power saving in the cache system. An analytical model for power estimation and average memory access time is presented in [2]. The method in [48] uses a two-step approach – first collects immediate data about the application using simulation, and then uses equations to predict the performance and power consumption of each of the possible configurations of the system parameters. An energy-aware mapping for tile-based NoC architectures under performance constraints is discussed in [53]. A system-level power aware design flow is proposed in [86] that shifts in design effort towards system-level in order to avoid failures after months of design time spent at Register Transfer Level (RTL)

and Gate Level. In [88], a technique to reduce the power consumption and latency in 2-D Mesh NoCs using globally pseudo-chronous locally synchronous clocking is proposed. The power management optimization is formulated using closed loop control concepts [118]. Each node has its local power manager that determines the power state. The estimator tracks changes in the system parameters and recalculates the new power management policy. Local power manager's interaction with other cores enables network-centric power management. Results show that large power saving is possible with good QoS. The Embedded Microprocessor Benchmark Consortium (EEMBC) develops a standardized method for simultaneously measuring power and performance on processor based systems [66]. Power consumption may significantly vary with different cache configurations and workloads. This method helps classify processors for the appropriate target markets (portable versus line-powered). A high-level simulation platform for multiprocessor systems is presented in [92]. Multiprocessing chips will have heterogeneous, programmable hardware elements that lead to different execution times for the same software executing on different resources as well as a mix of desktop-style and embedded-style software. They will also have a layer of programming across multiple programmable elements forming the basis of a new kind of programmable system which is referred as a Programmable Heterogeneous Multiprocessor (PHM). Current approaches using instruction set simulation for performance modeling of single-core systems would become far too prohibitive in terms of simulation time for the heterogeneous multiprocessing systems. They describe the foundations of a layered approach to modeling and performance simulation of PHM. They modeled a multi-processor SOC at high-level using MESH (Modeling Environment for Software and

Hardware) and low-level using cycle-accurate ISS (Instruction Set Simulator). Experimental results show only 5.5% of error in an average. The trade-off between the energy dissipation of software and that of system resources like cache and main memory is shown in [67]. It is concluded that there is no straight forward way to evaluate the change in total system energy for the changes in system parameters and applications. Optimization tools can be used to optimize various cache parameters.

Embedded avionics systems usually have caches and support pipelining for higher performance; they need offline guarantees for the satisfaction of their timing constraints. The variability of execution times exists on all system layers, in the processor architecture and the software development for single tasks. Approaches to improve the average case behavior of systems may be very unfortunate. Predictability of real-time systems is a lively research area. AbsInt's aiT tool is discussed in [143]; aiT is in routine use in the aeronautics and the automotive industries. In [68], an OS-controlled cache predictability mechanism for real-time systems is presented.

Our goal, in this work, is to implement a cache optimization methodology that improves the predictability and performance/power ratio at the same time.

2.4 Workload

The workload defines all possible scenarios and environmental conditions that the system-under-study will be operating under. The quality of the workload used in the simulation is important for the accuracy and completeness of the simulation results [40], [120]. A workload-based approach to performance testing that can be applied to large industrial software systems is described in [13].

To satisfy the future demands embedded systems should incorporate many more functions for real-time and background computations [146]. The performance, predictability, and power analysis of a real-time embedded system usually relies on the worst-case execution time (WCET) of the target applications. The traditional worst-case analysis may return overly pessimistic estimates of the system performance. This may cause catastrophic failure for multi-core real-time systems where each core may have its own OS and clock. A new method to characterize tasks with variable execution needs is presented in [77].

Video coding techniques from the Moving Picture Experts Group (MPEG) are described in [42], [165]. MPEG, a working group within the International Organization for Standardization (ISO), finalized MPEG4 (Part-2) in 1998. MPEG4 delivers professional-quality audio and video streams over a wide range of bandwidths over the Internet, from cellular phone to broadband and beyond. The development of H.264/AVC is summarized in [106], [112], [149], [171]. The Video Coding Experts Group (VCEG) from the ISO MPEG and the International Telecommunication Union's Telecommunication Standardization Sector (ITU-T) has developed a new standard, Advanced Video Coding (AVC) – widely known as H.264/AVC or MPEG4 Part-10. H.264/AVC is the improved version of H.263 and it is widely used in videoconferencing systems. We consider MPEG-4 and H.264/AVC applications in this work.

Detailed traces can be collected using ARMulator to drive the VisualSim simulation program [5], [156]. FFmpeg and JM-RS (96) are used with Cachegrind to decode MPEG4 and H.264/AVC encoded file [161], [164]. Workload is generated after post-processing Cachegrind output.

2.5 Simulation

Performance evaluation is a long-standing research topic [78], [79]. Various performance modeling techniques are discussed in [83], [91] as shown in Table 2.1. Direct measurement is a post-design step and not useful for systems under design. Analytical methods are okay for preliminary design explorations. However, analytical methods are not suitable for assessing detailed design trade-offs of future complex system architectures.

Table 2.1: Performance evaluation techniques

Measurement	**Analytical**	**Simulation**
Post-design step	Current/pre- design step	Pre-design step
SW monitor inside chip	Mathematical description	Computer programs
Accurate	Fast evaluation; less accurate	Slow evaluation; more accurate
Less flexible	Less flexible	More flexible

Simulation using computer programs and/or tools has become very popular in last decade [49]. The increased complexity of future architectures and applications make performance simulation very challenging. Understanding the performance of multiprocessors and distributed systems requires analyzing them separately and observing their interaction with the entire system architecture A Real-TIme System Simulation Tool (ARTISST) is described in [37].

Performance evaluation methodologies in articles appearing in the Proceedings of the International Symposium on Computer Architecture are shown in Table 2.2. Here, the total is not necessarily the sum across the columns because some papers used more than one evaluation method. This table is adapted from Reference [152].

Table 2.2: Performance evaluation methodologies used by researchers [152]

Year	Total Papers	Simulation		Measurement		Analytical		Others	
		Total	%	Total	%	Total	%	Total	%
2004	31	27	87	3	10	1	3	0	0
2001	25	22	88	2	8	0	0	2	8
1997	30	24	80	6	20	0	0	0	0
1993	32	23	72	9	28	6	19	1	3
1985	43	12	28	1	2	14	33	16	37
1973	28	2	7	0	0	5	18	21	75

Distributed simulation for chip-multiprocessors (CMP), introduced in [31], is based on message passing interface (MPI). This approach is found effective in distributed host system consisting of workstations connected with a high-speed network.

Various trace-driven methods and over 50 trace-driven simulation tools are surveyed in [135]. Considering the strengths and weaknesses of different approaches, no single method is the best when all criteria (including accuracy, speed, flexibility, expense, and ease-of-use) are considered.

We perform an extensive search for suitable tools to develop the simulation platform for our usages. In [107], some modeling and simulation languages and tools are listed with a short description for each item. BuildSim, LabView, MicroSaint, SimCreator, SimuLink and MatLab, VisSim, and VisualSim are some of the popular simulation tools. We install and test some of them in FAU Mobile Computing Laboratory and 319 Open and Teaching Laboratory. We find that no single tool is suitable enough for real-time multi-core system analysis. We select a set of simulation languages and tools consisting C, VisualSim [172], Heptane [162], and Cachegrind [159] (with FFmpeg [161] and JM-SR (96) codec [164]) to develop and run our simulation programs.

2.6 Other Related Issues

In the previous subsections, we have discussed some selected articles focusing on cache optimization, embedded system, and performance-power- predictability issues. We have also summarized some articles addressing the workload and simulation issues. In this subsection, we cover other related issues including the execution time predictability, performance/power ratio, and scalability in real-time embedded systems.

In [8], [10], [74], and [136], the architecture of an embedded system running multimedia applications is explored. A simulation program is developed to evaluate the system performance in terms of utilization, delay, and total transactions for various CL1 sizes. In [8], the memory latency of cluster-based cache-coherent multiprocessor systems with different interconnection topologies is evaluated. In [10], cache parameters are fine tuned for embedded MPEG4 and H.264/AVC video CODEC.

The demand for supporting real-time applications in embedded systems is growing. Execution time predictability is important for any real-time system. Cache improves performance but poses challenges by increasing the unpredictability. It has been shown that for embedded systems with a well known workload, static cache locking helps to determine the worst case execution time (WCET) and cache-related preemption delay. In [11], a static I-Cache locking scheme is proposed that makes the real-time embedded system more predictable. FFT, MI, and DFT applications are used as inputs to run the simulation program. CPU utilization for both static cache analysis (no cache-locking) and cache locking is obtained by using the Heptane tool. Experimental results show that the cache locking scheme improves both predictability and performance when the right cache

blocks are locked and appropriate cache parameters are used. Experimental results also show that predictability can be further enhanced by sacrificing the performance.

A data bypass cache and a predictor table with CL1 and CL2 (as victim cache) are used in [75]. This method reduces access latencies by determining whether a load should bypass the main cache hierarchy and issue an early load to main memory. This technique improves performance and power.

One of the biggest challenges in shared memory multi-core (or multi-processor) systems is scalability. In [65], a group of researchers from Stanford University propose DASH (Directory Architecture for SHared memory) architecture, where snoopy protocol is used inside the cluster and clusters are connected via a mesh interconnection network. The DASH architecture achieves near-linear performance growth as the number of processors increases from a few to a few thousand. The architecture significantly reduces the memory latency and provides higher processor utilization and higher overall performance [41]. In [74], the DASH architecture is enhanced by adding CL2. Ring and hypercube networks are evaluated in addition to the mesh network. The architecture is simulated for a total of 16 processors – 2 or 4 processors in a cluster. Simulation results show reduced memory latency due to the addition of CL2.

It is important to note that Moore's law encourages multi-core embedded systems; however the performance does not depend only on the number of cores. According to the law of diminishing returns, in a production system with fixed (say, size) and variable (say, labor) inputs, beyond some point (size and labor), each additional unit of variable input produces less and less additional output. From Amdahl's law by Gene Amdahl, the speedup of parallel computing is limited by the fraction of the problem that must be

performed sequentially [61]. So the scalability should be justified against the performance gain (that depends on architecture, system software, and applications).

2.7 Summary

The hunger for faster performance is never satisfied. Every new performance advance in processors leads to another level of greater performance demands from consumers. Most manufacturers are adopting multi-core processors to acquire additional processing speed for very little additional power consumption.

Important advantages and disadvantages of multi-core processors are summarized in Table 2.3. On one hand, multi-core processors improve performance for a very little extra energy; multi-core architectures take less die area and provide high cache/bus snoop performance. On the other hand, multi-core systems require higher bus/memory bandwidth. Some multi-core architecture requires new software. Existing application codes running in single-core systems need to be re-written to take advantage of multi-core architecture.

Table 2.3: Advantage (+) and disadvantage (-) of multi-core processors

Item	**Advantage (+) or Disadvantage (-)**
Performance	+
Power consumption	+
Die area	+
Cache/bus snoop performance	+
Bus/memory bandwidth	-
Existing software usages (Cell processor: in-order, no branch prediction, etc)	-
Real-time support / predictability	??

Multi-core processors bring high processing speed for trivial additional power, as well as the "multi-core cache unpredictability". If the unpredictability is not manageable, it may cause catastrophic failure of the entire system. In order to improve predictability, IBM is not using pipelining in Cell and cache in SPEs, but that makes software development more complicated and may cause shared level-2 cache bottleneck for larger applications. Studies show most vendors including Intel and AMD are using level-1 and level-2 caches in their multi-core processors.

We know that cache is the primary source of execution time unpredictability. Real-time systems are currently not taking full advantages of cache memory especially in multi-core environment. Research needs to be done to develop methodologies so that multi-core processors can improve both predictability and performance to support real-time systems consuming no or very little additional power. In the following chapters, we present various cache modeling and optimization techniques to improve the execution time predictability and performance/power ratio of multi-core, as well as single-core, real-time embedded systems.

CHAPTER 3

CACHE MODELING AND OPTIMIZATION FOR SINGLE-CORE SYSTEMS

The popularity of embedded systems in various fields such as multimedia, signal and image processing, high-speed networks, and information systems is on the rise. To satisfy the growing consumer demands, more functions are being added to modern computing systems. With improved CPU speed, cache memory sub-system deficiency is the major barrier to improving the performance of such complex systems. Studies show that there is sufficient reuse of values for caching that should significantly reduce the memory bandwidth requirement for real-time applications. Each application has its own composition of the instruction and data set. Proper understanding of the cache memory hierarchy and the target application is obvious to improve the performance of such a system. The focus of this chapter is cache modeling and optimization for single-core systems to enhance performance. The architecture we simulate includes a cache memory subsystem with two levels of caches. We use VisualSim and Cachegrind simulation tools to model, analyze, and optimize cache size, line size, associativity level, and cache levels of the target architecture running H.264/AVC video decoding algorithm.

3.1 Introduction

Applications drive the architecture of embedded computing systems. For multimedia processing, low-power high-performance adaptive computing architecture is expected [111]. For many embedded devices, single-core implementation is recommended to meet the power consumption and cost requirements. Due to the improvements in the semi-conductor industry, required computational speed to implement such a device is easy to achieve. However, due to the growing disparity between the increasing CPU speed and the memory speed, the amount of traffic from CPU to memory leads to a significant processor/memory speed gap. A small, fast, and expensive memory, called cache, is used between CPU and main memory to improve performance by dealing with memory bandwidth bottlenecks and other issues like bus contention problem.

The basic cache memory organization of a single-core architecture is shown in Figure 3.1. As shown in Figure 3.1(a), if CPU speed is 400 MHz and main memory speed is 100 MHz (memory is slower than the CPU), then CPU wastes at least 3 cycles if it needs something from main memory. Cache improves performance by reducing the data access time. Using data stored in the cache helps cut down bus traffic significantly. As shown is Figure 3.1(b), first-level or on-chip cache (CL1) is on the same chip with the CPU and operates almost at the same speed as the CPU does. CL1 may be split into level-1 instruction cache (I1) and data cache (D1) for improved performance. Due to the success with CL1, second-level or off-chip cache (CL2) has been introduced (see Figure 3.1(c)). Today, multi-level caches in various configurations are available. Data transfer between CPU and cache and cache and main memory are different. Data between CPU

and cache is transferred as data object and between cache and main memory as block as shown in Figure 3.1(d).

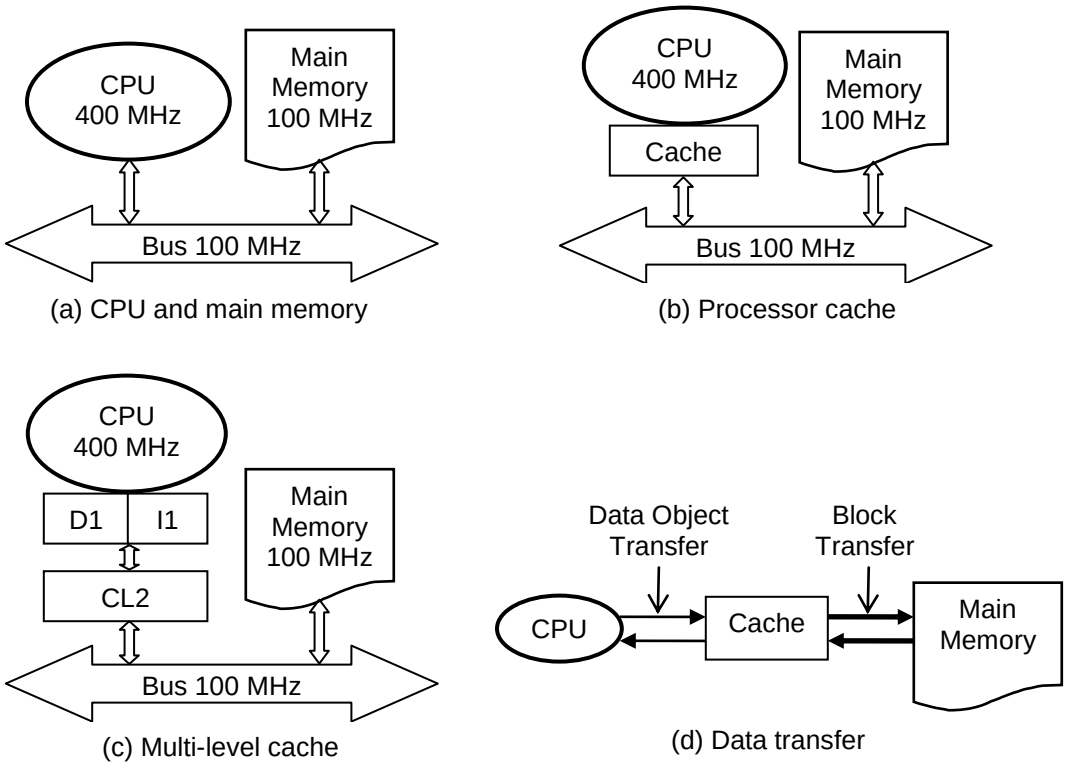


Figure 3.1: Cache memory hierarchy – cache, main memory, and data transfer.

3.2 Performance Improvement

Modeling and simulation technique is very effective for the early analysis of any computing system. For future products (when the actual product does not exist yet), conceptual modeling and simulation is probably the only way to make scientific design decisions. As we already know, most single-core architecture has cache to improve performance. However, cache performance (and the overall system performance) for a target application greatly depends on the values used for various cache parameters (like cache size, line size, and associativity level). Larger cache size usually improves

performance by reducing capacity cache miss. But, cache is expensive and after a limit performance improvement for increased cache size is not significant. Bigger line size helps improve performance by reducing compulsory cache miss. Conversely, aggressive increment in line size introduces cache pollution and decreases performance. Higher associativity level is proven to improve performance by reducing conflict cache misses. In spite of this, after a point increase in associativity level makes the system more complicated without any significant performance improvement. Therefore, it would be beneficial to analyze and optimize the cache memory hierarchy in a single-core embedded system using the target applications.

3.3 Modeling and Simulation

In this chapter, we model a single-core architecture and simulate the model to improve the performance by tuning various cache parameters. Simulation details are presented in the following subsections.

3.3.1 Assumptions

In order to simplify the model and run the simulation program, we make the following assumptions.

- A single-core system with CL1 (I1 and D1) and CL2 is considered.
- The dedicated bus that connects CL1 and CL2 introduces negligible delay compared to the delay introduced by the system bus which connects CL2 and main memory.

- Task time has been divided among CPU, CL1, CL2 (when exists), bus, and main memory proportionally.
- Write-back update policy is implemented for reduced CPU utilization. According to this policy the CPU is released immediately after CL1 is updated.
- CIF YUV 4:2:0 formatted (352 pixels by 288 lines, 30 fps) video stream file has been used to generate a representative .264 file.

3.3.2 Simulated Architecture

The target architecture has a processing core to run video decoding algorithm. The system has CL1 and CL2 caches. CL1 is a split into I1 and D1 caches and CL2 is a unified cache as shown in Figure 3.2. CL2 and the main memory are connected to a shared bus. The processing core communicates with the main memory via CL1 and CL2. The compressed (i.e., encoded) video files are in the main memory. The core reads them using the cache memory subsystem, processes (decodes) them, and writes them back to the main memory through the caches.

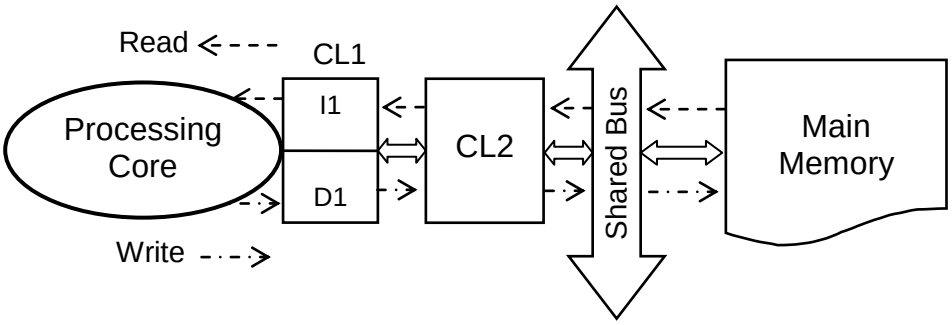


Figure 3.2: Simulated architecture for embedded system running H.264/AVC decoder.

3.3.3 Simulation Tools and Workload

In this chapter, we use trace-driven hardware/software co-simulation technique. Using VisualSim we develop the model of the target architecture and run the simulation program. We use workload of H.264/AVC decoding algorithm, an important and demanding multimedia application, to run our simulation program. Using Cachegrind and JM RS (96), we collect I1, D1, and CL2 miss rates and D1 and CL2 read/write references for various cache parameters. We evaluate the performance in terms of miss ratio and total number of transactions processed by different system components for various cache size, line size, associativity, and cache levels.

The original YUV file we use to generate .264 file is a common intermediate format (CIF) YUV 4:2:0 video stream with image format 352 pixels by 288 lines, 30 fps. Using H.264/AVC JM RS (96) CODEC, we decode a .264 file (to a new YUV file). Using Cachegrind, we collect the total references for I1, D1, and CL2. Data (D1) and CL2 references consist of read and write references. As shown in Table 3.1, the H.264/AVC file (.264 file) that we decode has 62% instruction and 38% data references at CL1. Data references are consist of 75% read and 25% write. At CL2, we see 84% read and 16% write references. These numbers are used to run the simulation program.

Table 3.1: Workload – CL1 (I1 and D1) and CL2 references while decoding .264 file

Total CL1 References (K)	**CL1 References (K)**		**D1 Refs (K)**	
	I1	**D1**	**Read**	**Write**
194322	120163 (62%)	74159 (38%)	55765 (75%)	18394 (25%)
Total CL2 References (K)			**CL2 Refs (K)**	
			Read	**Write**
191			161 (84%)	30 (16%)

3.3.4 Experimental Results

In this chapter, we model and optimize the cache memory subsystem of a single-core architecture running H.264/AVC video decoding algorithm to improve performance. We present the results obtained using Cachegrind and VisualSim in this subsection.

First, we present the miss ratio due to the variation of CL1 (I1 + D1) cache size where CL1 line size is fixed at 32B and associativity at 8-way are shown in Figure 3.3. We keep, CL2 fixed at cache size 1M, line size at 32B, and associativity at 8-way. Experimental results show that the miss rate decreases when CL1 size is increased from 8K + 8K to 16K + 16K. It is also observed that the decrease in miss rate of D1 is higher than that of I1. According to our simulation results, for CL1 (D1 + I1) size 16K + 16K and bigger, the decrease is not significant. So using a CL1 (I1 + D1) size greater than 16K + 16K may not offer any benefit.

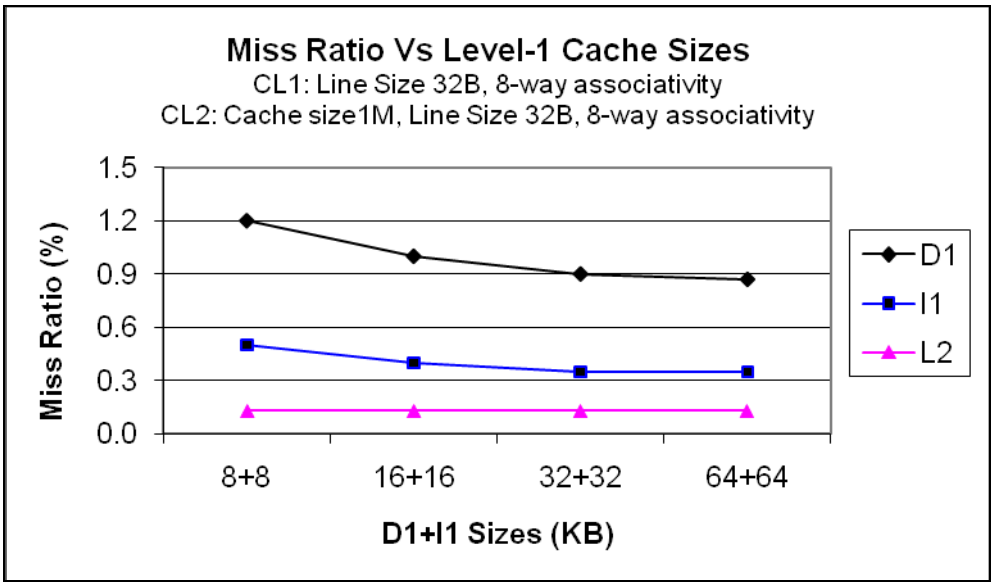


Figure 3.3: Miss ratio versus CL1 (I1+D1) size.

Second, the miss ratio due to the variation of CL2 cache size where CL2 line size is fixed at 32B and associativity at 8-way are shown in Figure 3.4. In this case, CL1 is fixed at cache size 16K+16K, line size 32B, and associativity 8-way. It is observed that for CL2 size of 256K or smaller the miss rates remains unchanged, from 256K to 2M the miss rates decrease sharply, and for 2M or bigger the miss rates decrease slowly until it becomes 0%. From cost, space, and complexity standpoints, larger CL2 (bigger than 2MB in this case) may not provide any significant advantage.

We obtain the miss rates for various CL2 size using both Cachegrind and VisualSim (see Figure 3.4). Results from VisualSim and Cachegrind follow the same pattern supporting the fact that for CL2 size from 256 KB to 2 MB the miss ratio decreases sharply. For other CL2 sizes, the decrease in miss ratio is negligible.

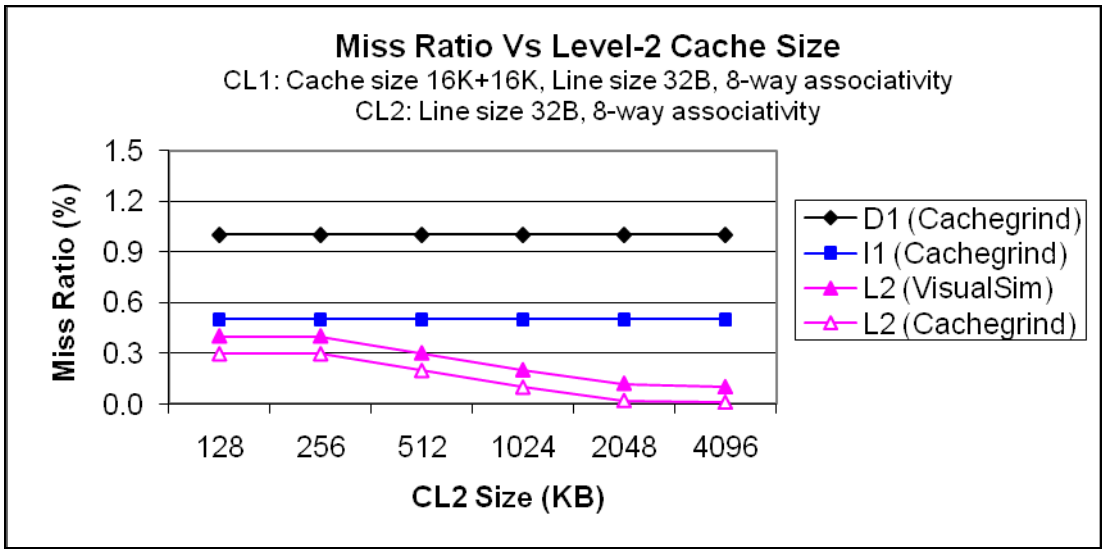


Figure 3.4: Miss ratio versus CL2 size from VisualSim and Cachegrind.

Third, increasing the line size reduces compulsory cache miss ratio (compulsory miss occurs on the first access to a block). However, too large a line size may increase capacity cache misses (capacity miss occurs because blocks are discarded from the cache as the cache cannot contain all blocks needed for program execution). Figure 3.5 shows that for a smaller cache size, miss rates start decreasing with the increase in line size. After a certain point, known as cache pollution point (128B for D1 in this case), miss rates start increasing. In this case, line size 128B or higher may not be efficient as it requires more data to be read and written (in case of a miss).

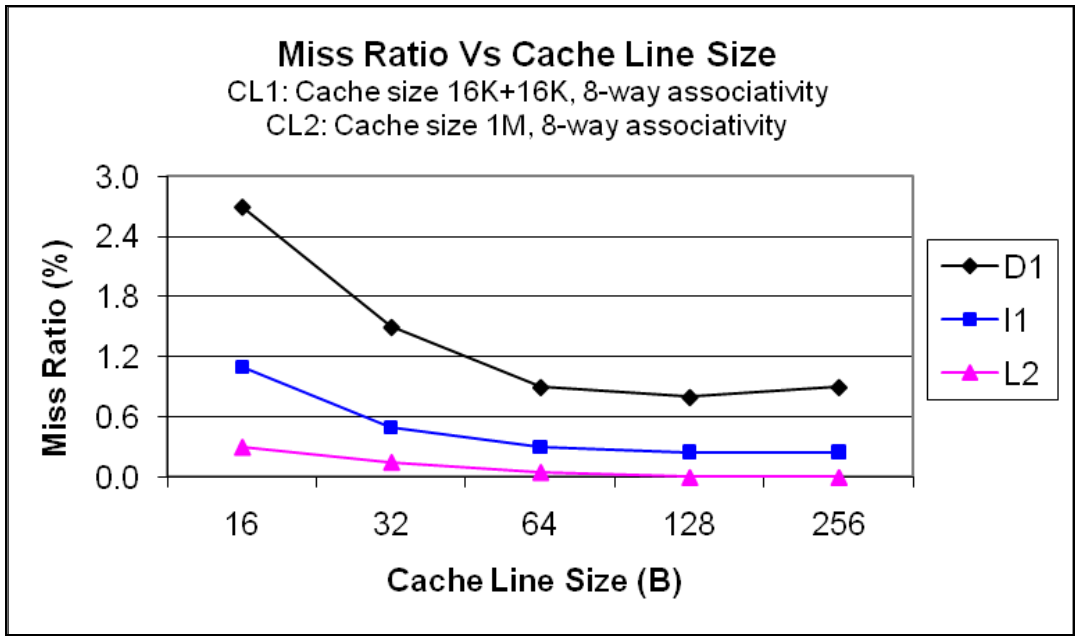


Figure 3.5: Miss ratio versus line size.

Fourth, using Cachegrind, we collect miss ratio by varying associativity level for CL1 (D1+I1) cache size 16K+16K, CL2 size 1M, and line size 32B. The miss rates for different levels of associativity are shown in Figure 3.6. The miss rates significantly decrease when we go from 2-way to 4-way associativity. Going to 8-way to higher, the changes are not significant.

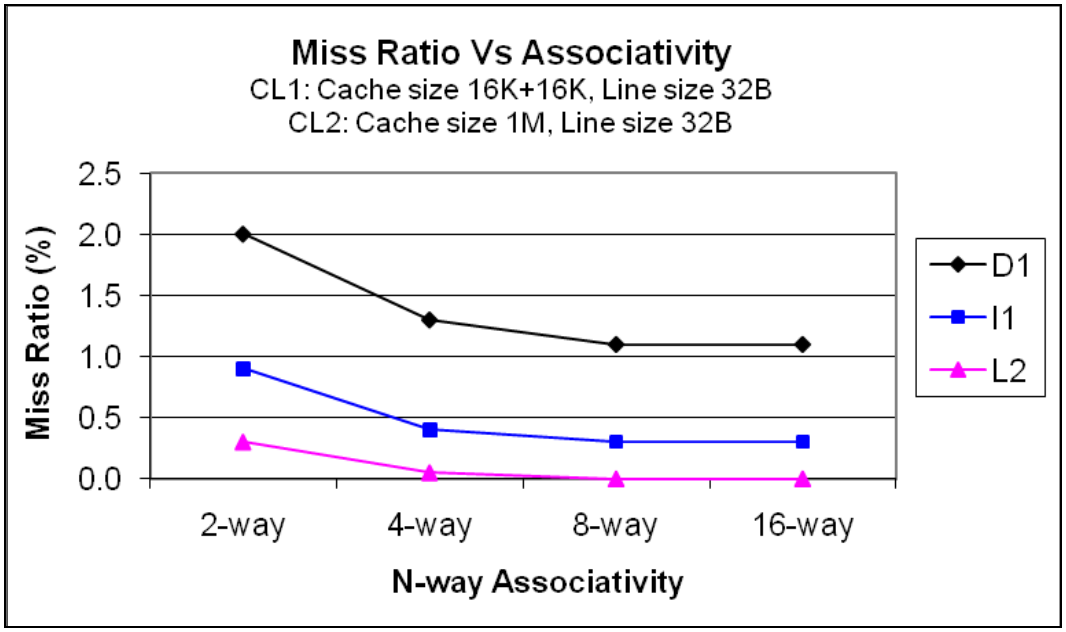


Figure 3.6: Miss ratio versus associativity.

Using VisualSim, we investigate the impact of the presence of a CL2. CL2 size is varied and performance metrics, namely total number of transaction and CPU utilization, are obtained. We keep CL1 (D1+I1) size fixed at 16K+16K, line size at 32B, and associativity at 8-way. We change CL2 size from 128K to 4 MB and keep CL2 line size fixed at 32B and associativity at 8-way.

In our simulation, memory references are initiated at the CPU and are referred to CL1; if not satisfied in CL1, is referred to CL2. Finally, requests unsuccessful at CL2 are satisfied from the main memory (MM) via the shared bus. Simulation results show that total number of transactions through the bus and main memory decrease with the increase of CL2 size as shown in Figure 3.7. We run our simulation program for a total of 1M transactions; 380,000 of those transactions are data and the rest 620,000 are instructions. Transactions are referred to D1 or I1 by the CPU. For D1 miss ratio 1.0% and I1 miss ratio 0.4%, (3,800 + 2,480) or 6280 requests go to CL2. For CL2 size 128K (miss ratio is 0.3%), only 19 requests go to MM via the shared bus. For CL2 size 2M (miss ratio is 0.03%) only 2 tasks go to MM. For CL2 cache size 4M or higher, no requests go to the main memory as miss ratio becomes 0.0%.

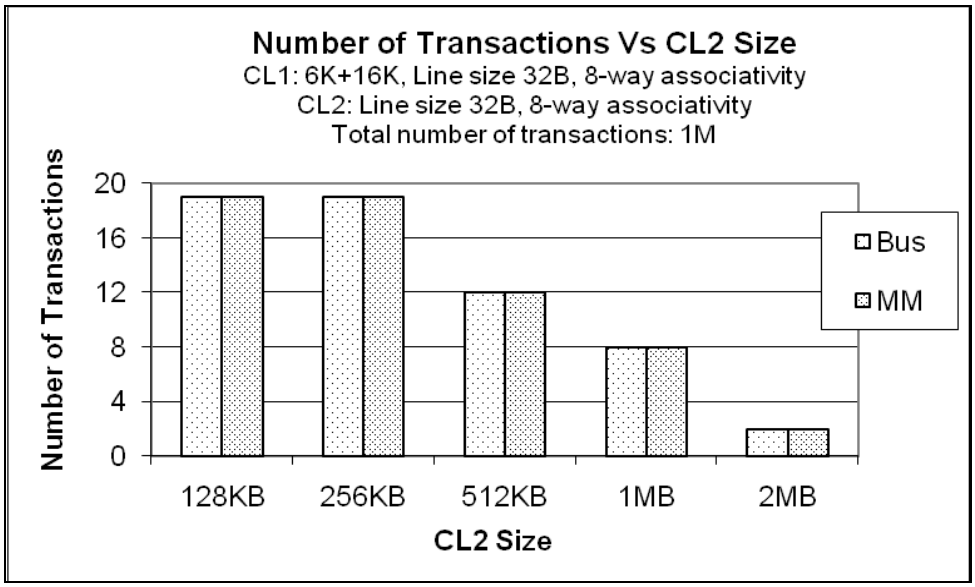


Figure 3.7: Total number of transactions versus CL2 size.

Studies show that addition of CL2 decreases the bus traffic and latency and improves system performance.

Using VisualSim simulation model, we investigate the impact of various video files with different bit-rates on CPU utilization. Bit-rates may be different for various reasons including quality of service and index of animation (example, a landscape panning is coded differently from a football match). In our simulation, we keep CL2 fixed. We start with task-rate 0.1 tasks per unit time and increase it up to 2 tasks per unit time. Simulation results show that utilization increases with increased task-rate. At task rate of 2 tasks per simulation time units, utilization is close to 100% for CL1 (D1+I1) size (8K + 8K) as shown in Figure 3.8. Further increase in task rate (with other parameters unchanged) causes the system to break down. System cannot process all the tasks, once the task queue is full it starts dropping the tasks. A bigger cache size at this situation should reduce the utilization. In this experiment, CL1 (D1+I1) size (64K + 64K) reduces CPU utilization by 30% (approximately).

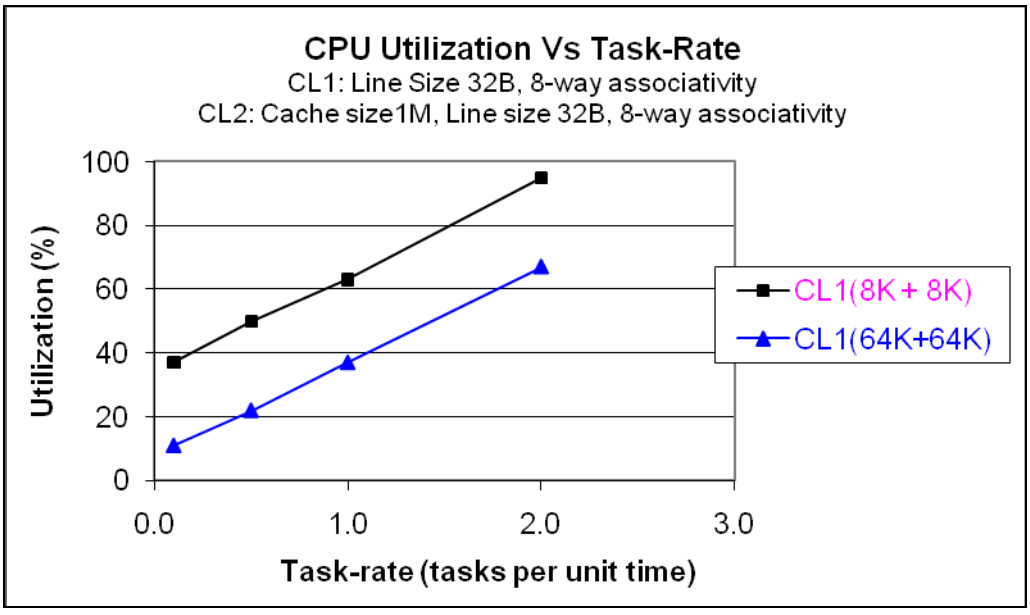


Figure 3.8: CPU utilization versus task-rate.

3.4 Summary

In this chapter, we focus on enhancing the performance of a single-core embedded system through cache modeling and optimization for H.264/AVC decoder. The architecture includes a processor to run the decoding algorithm and a two-level cache memory subsystem. We optimize cache size, line size, associativity level, and cache levels for the system. Simulation results show that this cache modeling and optimization technique can be effectively used to enhance the performance of single-core embedded systems running real-time applications.

CHAPTER 4

CACHE MODELING AND OPTIMIZATION FOR MULTI-CORE SYSTEMS

Power consumption and dissipation are important design factors for modern computing systems. Excessive power consumption is a direct threat to the battery-life of a battery operated device. High power dissipation requires a sophisticated cooling system. From die area and cost point of view, power consumption is very crucial for embedded systems. In Chapter 3, we have seen how cache modeling and optimization helps improve performance of single-core systems. In this chapter, we improve performance/power ratio of multi-core embedded systems by cache optimization. Both single-core and multi-core embedded systems have caches. In multi-core systems, the cache memory subsystem is more complicated than in single-core systems. Even though multi-core architecture improves performance/power ratio by simultaneously processing the tasks and running at a lower frequency, there are opportunities to improve performance/power ratio of this architecture by cache optimization. We simulate a dual-core system and run the simulation program using MPEG4 decoding algorithm. Experimental results show that performance/power ratio can be improved by optimizing cache parameters. Results also show that CPU utilization and total number of transactions can be adjusted by changing the cache size and task rate.

4.1 Introduction

Multi-core processor is the future of computing as it improves performance/power ratio. Embedded systems increasingly use multi-core processors in order to satisfy the computational needs. In a multi-core CPU or chip-level multiprocessor (CMP), two or more independent cores are combined into a die. There may be multiple levels of caches and caches may be organized in a number of ways. The processors share the same interconnect network with the rest of the system. Key hardware components for such a system include – CPU, memory, bus, cache, etc. Other important components are operating system and applications. Such a system may be homogeneous (distributed system contains same kind of hardware and software) or heterogeneous (distributed system contains many different kinds of hardware and software working together in cooperative fashion to solve problems). Figure 4.1 shows a multi-core system. In a multi-core system, the applications are broken into smaller tasks and given to different processors. A processor may be designed to perform a specific task assigned to it. For example, CPU-1 runs on an open operating system, has only CL1, and performs Task-1. On the other hand, CPU-N has CL1, CL2, CL3, and real time operating system (RTOS) to perform Task-N. All CPUs share main memory through a single shared bus.

In this chapter, we consider a simplified multi-core architecture with 2 processing cores; each core has its own private CL1. We develop a cache modeling platform for this architecture and optimize cache parameters to analyze and improve performance and decrease power consumption (i.e., improve performance/power ratio).

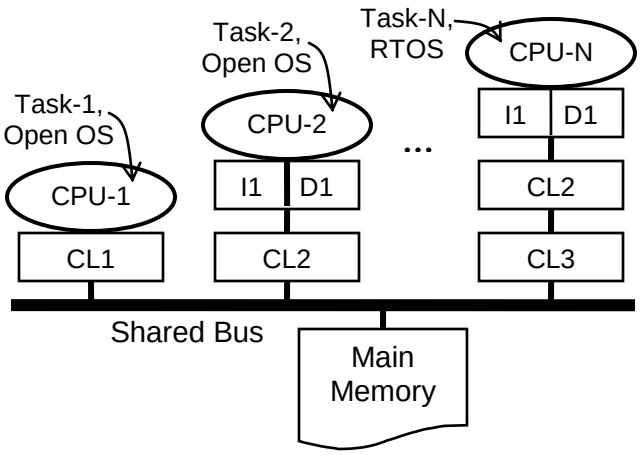


Figure 4.1: Shared bus multi-processor architecture.

4.2 Performance/Power Ratio Improvement

If cache parameters are not properly chosen the system may operate under poor performance and may consume huge amount of power. Like single-core, multi-core performance/power ratio can be improved by cache optimization. However, cache optimization in multi-core is difficult. In multi-core, each core may have its own private CL1 and there may be private and/or shared CL2 (and higher levels of caches). Cache optimization in multi-core systems is very important to improve performance/power ratio. This cache optimization technique is very effective for homogeneous multi-core system as each core should be equally capable of processing tasks.

4.3 Modeling and Simulation

We simulate a dual-core embedded system to improve the performance/power ratio by cache optimization. Simulation details are presented in the following subsections.

4.3.1 Assumptions

The following assumptions are made to simulate cache optimization in a multi-core architecture.

- A real-time embedded system with two cores is considered in order to evaluate the performance/power ratio accurately. When both cores are used, one of the two cores, DSP (a digital signal processor), processes video data in real-time and the other core, AP (an application processor), displays the processed video data.
- AP (runs under general purpose operating system) is capable of handling the traffic in sync with DSP (runs under real-time operating system).
- DSP and AP use IPC/MU to communicate with each other without any delay.
- Data transfer between CPU and cache is considered as a frame.
- Size of shared main memory is unlimited – there is no delay involved with the actions of reading from buffer, writing into main memory, etc.

4.3.2 Simulated Architecture

To analyze the impact of cache optimization on performance and power consumption of a multi-core embedded system, we simulate an architecture that has two processors. The architecture is designed to support video communication applications. When both cores are used, one of the two cores, a digital signal processor (DSP), decodes the encoded video streams and the other core, an application processor (AP), plays it back. Both DSP and AP have CL1. As illustrated in Figure 4.2, DSP reads and writes video data via its cache and AP reads the decoded video data using its cache. DSP and

AP use inter-processor communication (IPC) with a register-based messaging unit (MU) and a shared memory system. Here, main memory is being shared by DSP and AP and they are connected via a shared bus. DSP writes the decoded video streams into the main memory and sends a message to AP. AP reads the video streams and plays them back.

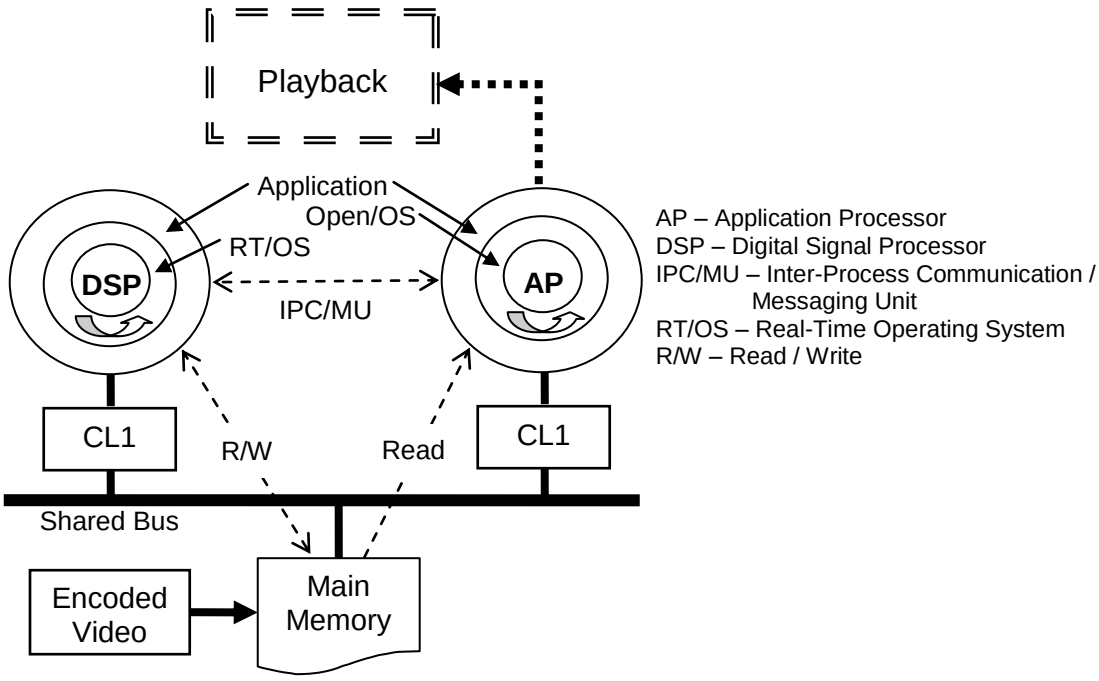


Figure 4.2: Simulated architecture of a dual-processor system.

4.3.3 Simulation Tools and Workload

We use VisualSim to model the architecture and run the simulation program. VisualSim provides functionality blocks to model multi-core embedded systems and real-time applications. MPEG4 video decoding algorithm is used to run the simulation program. The video file is formatted with Common Intermediate Format (CIF), YUV 4:2:0, width 352 pixels, height 288 lines, and 30 frames per wall-clock second. All three pictures types, namely intra-coded (I frame), predictive picture (P frame), and bi-

directionally predictive picture (B frame), are considered. The dependencies among frames are shown in Figure 4.3. For an example, P frames can be decoded from previous I (or P) frame. Both previous I (or P) and next P frames are needed to decode any B frame. For this study, we stay on the conservative side by assuming that the data transfer rate between CPU and the cache is in frames.

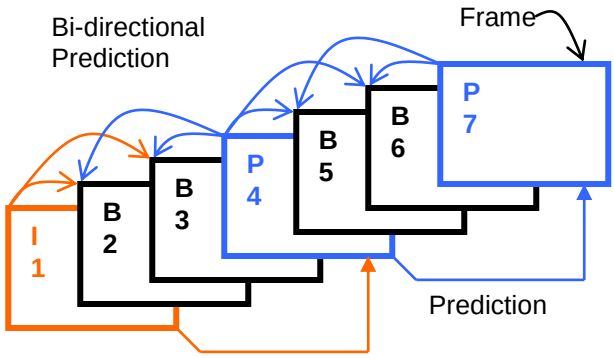


Figure 4.3: Sample MPEG4 GOP with 7 picture frames.

For DSP, the hit ratio and miss ratio are calculated based on the cache sizes and the MPEG4 decoding algorithm. Cache size is finite and fixed for a specific architecture. Frame size is also finite and fixed for a specific application/algorithm. The AP cache is assumed to use a pre-fetching scheme that in the event of a miss will pre-fetch the maximum number of frames that the cache can fit. So, if the AP cache size is N, then there is one miss per N frames and the hit ratio is (N-1)/N and the miss ratio is 1/N.

Total number of bytes (B) in a MPEG4 CIF YUV 4:2:0 encoded file is N*3*width*height/2. Here N is the total number of frames. So, for CIF YUV 4:2:0 352 pixels by 288 lines encoded video stream,

Frame Size = 3*352*288/2 bytes ≈ 152 KB

Table 4.1 summarizes the hit ratio and miss ratio of different CL1 sizes for both DSP and AP processor.

Table 4.1: Hit ratio and miss ratio for DSP and AP caches

Cache size (KB)	Max. Frames Cache can hold	DSP		AP	
		Hit (%)	Miss (%)	Hit (%)	Miss (%)
384	2	28	72	50	50
512	3	86	14	67	33
1024	6	86	14	84	16

From above table, DSP does not take advantage of increasing cache size from 512 KB to 1024 KB, so the hit ratio (also the miss ratio) remains unchanged for DSP. But the cache hit ratio of AP increases from 67 to 84 as cache size increases from 512 KB to 1024 KB. This is because to decode a B frame, DSP may need access to at most 2 other frames (previous I or P frame and the next P frame). So having more than 3 frames into the cache does not improve hit ratio over having exactly 3 frames into the cache for DSP.

4.3.4 Experimental Results

It is important to design the components of cost sensitive products for the worst-case but not to over-design it. In this work, we explore the impact of cache optimization of performance/power ratio of a multi-core embedded system running MPEG4 application. We stay on the conservative side by assuming that the data transfer rate between CPU and cache is in frames while generate workload to run the simulation program.

We first present CPU utilization for various cache sizes. Figure 4.4 shows CPU utilization for various cache sizes for task rate 0.1 time units. For cache size of 384 KB, we see that only two frames can be kept into the cache at a certain point of time. We know B frames are predicted based on both previous I (or P) frame and next P frame. As a result there would be more cache misses. When cache size is increased to 512 KB, three frames fitted into the caches at the same time and the miss rate is reduced significantly. Then we increase cache size to 1024 KB. It is noticeable that utilization of AP, bus, and memory dropped considerably. But DSP utilization did not change. This is because, even though DSP cache can have 6 frames at the same time, it can use at most 3.

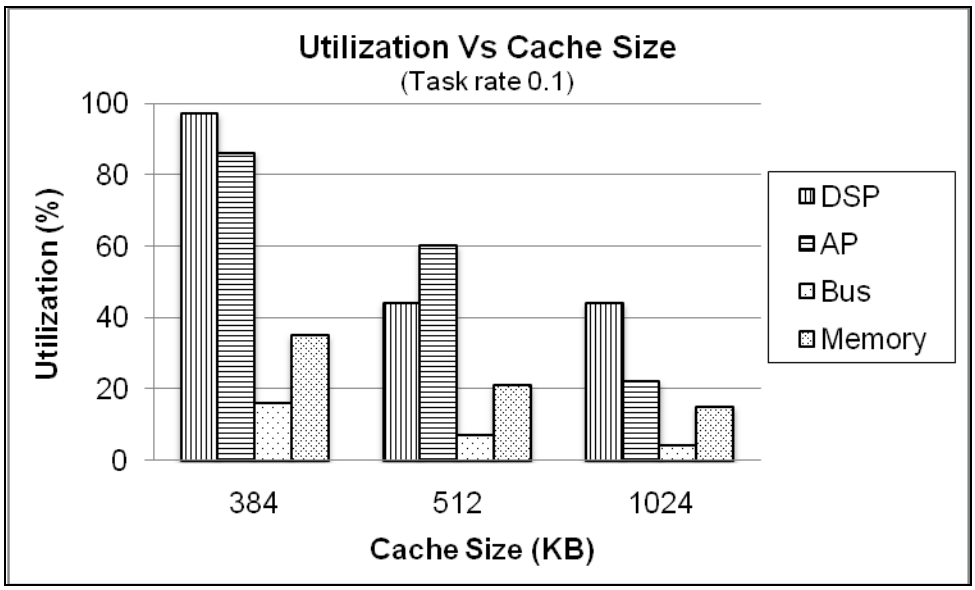


Figure 4.4: Utilization versus cache size with task rate 0.1 time units.

Figure 4.5 shows the results for task rate 0.2 time units. At this increased task rate, DSP utilization for cache size of 384 KB should go beyond 100% which is not possible. As a result simulation stops abnormally indicating a system failure. Please note that results for cache size 384 KB is not included in Figure 4.5. For cache size of 512 KB, AP utilization becomes extremely high. At this task rate, if cache size is increased to 1024 KB, AP utilization reduces significantly, but DSP utilization remains unchanged due to the fact that DSP does not take advantage of having more than 3 frames into the cache at the same time.

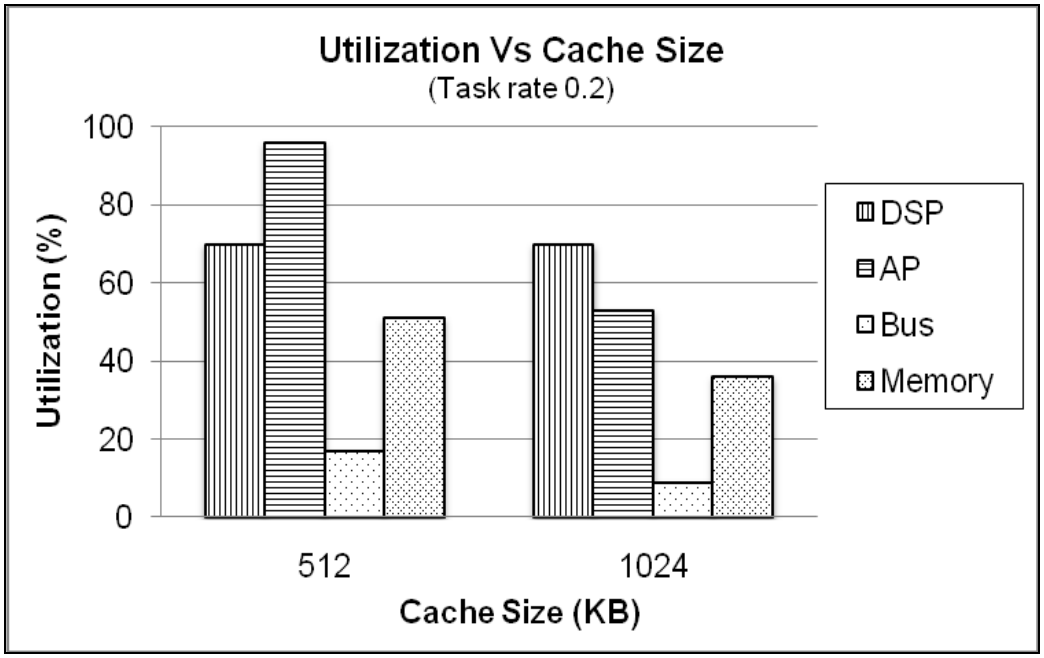


Figure 4.5: Utilization versus cache size with task rate 0.2 time units.

We also study mean delay for various cache sizes. For DSP and AP, mean delay is significant for cache size of 384 KB. As cache sizes increase, DSP and AP delay decreases. Bus and memory delay do not change significantly, since each of them processes almost the same number of requests almost at the same processing speed. It is important to notice that DSP mean delay remains the same for 512 KB and 1024 KB even though AP mean delay decreases.

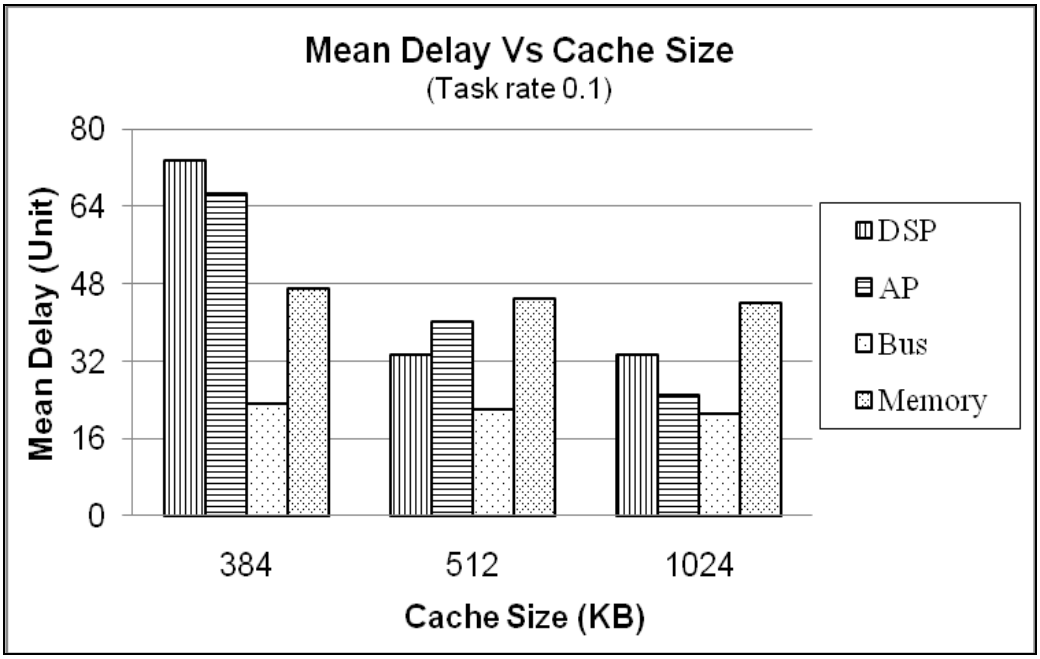


Figure 4.6: Mean-delay versus cache size.

Total power consumption for different cache sizes is shown in Figure 4.7. DSP power consumption decreases sharply from 384 KB to 512KB but remains unchanged from 512KB to 1024KB. However, AP power consumption decreases from 384 KB to 512KB and from 512KB to 1024KB. The differences between the total power consumption by the bus and main memory due to cache size changes are not significant, since the application has a very high hit ratio for all the cache sizes.

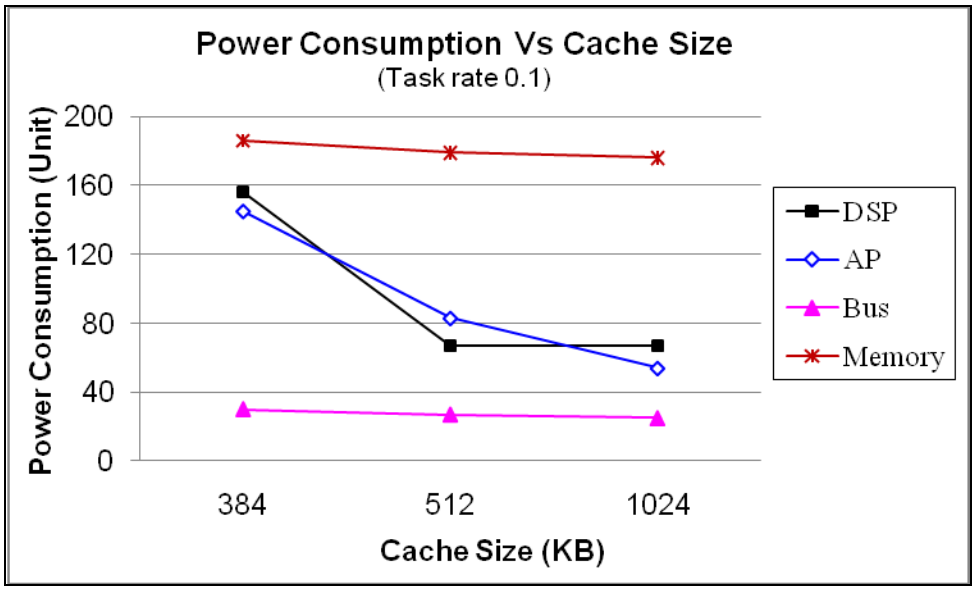


Figure 4.7: Total power consumption versus cache size.

We also investigate the impact of various cache sizes on the total number of transactions processed (see Figure 4.8). We measure transaction as the total number of tasks entered into and tasks exited from the component during the whole period of simulation. For DSP and AP, the total number of transactions processed remains unchanged with the variation of cache sizes. As cache size increases, more requests are

satisfied from the cache; as a result, total number of transactions for the bus and main memory decreases.

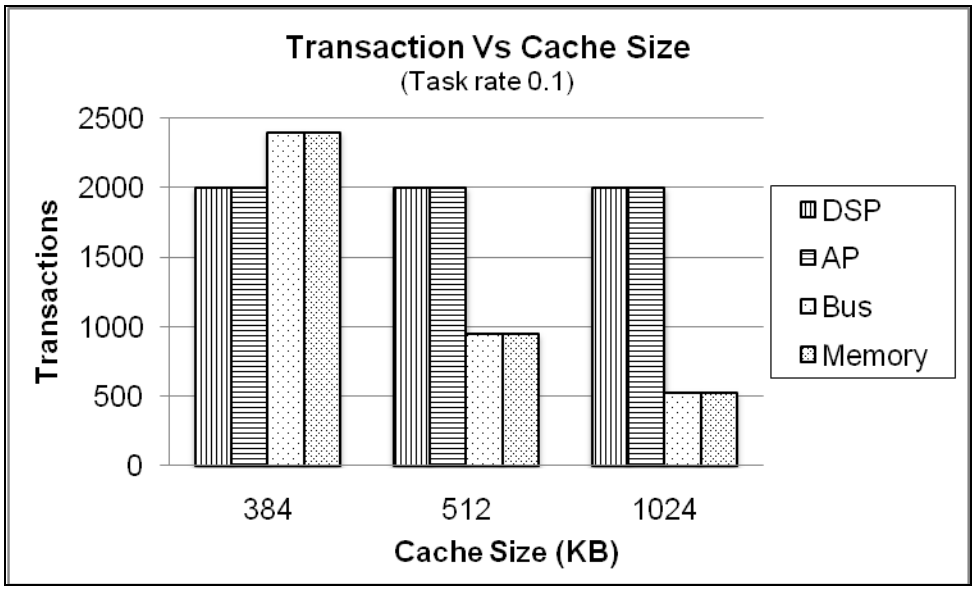


Figure 4.8: Transactions (tasks entered and exited) versus cache size.

From Figures 4.4 and 4.6, we observe that for cache size of 384 KB, DSP and AP utilization and delay are very high when compared with those for cache size of 1 MB. So for our architecture cache size of 512 KB is optimal where DSP and AP utilization are between 25% and 60% and the bus and main memory utilization are reasonable.

Finally, we investigate the impact of adding an additional core to a single-core system by comparing the mean delay and total power consumption by the single-core system and the dual-core system, respectively. In Table 4.2, the row with 1 core represents the results for a single-core and the row with 2 cores represents the results for a dual-core. Experimental results using cache size 384 KB and MPEG4 decoding workload show that total number of transactions per core becomes just half as the second

core is used. It is also observed that the mean delay decreased more than 45% for less than 5% additional power consumption.

Table 4.2: Performance Vs power consumption for single- and dual-core

Cache 384 KB			
Core(s)	**Performance**		**Total Power Consumption (unit)**
	Total Mean Delay (unit)	**Transactions Per Core**	
1	383	1000	858
2	210	500	901

4.4 Summary

In this chapter, we explore a cache modeling and optimization technique to improve performance and decrease power consumption of a multi-core embedded system running MPEG4 decoding algorithm. Experimental results show that adding more processing core improves performance for small amount of extra power (i.e., increases performance/power ratio). Simulation results, also, show that cache size and task rate have significant influence on CPU utilization. CPU utilization can be adjusted by changing cache size and task rate. This cache modeling and optimization technique can enhance the performance/power ratio of real-time multi-core embedded systems.

CHAPTER 5

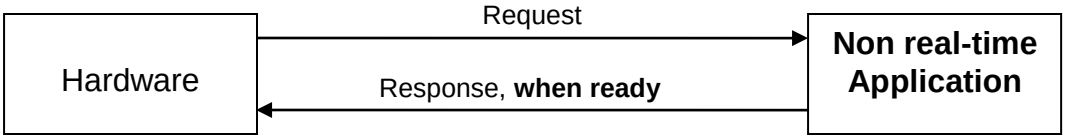
CACHE LOCKING TO IMPROVE PREDICTABILITY

Cache improves performance by reducing the speed gap between the main memory and CPU. However, the execution time becomes unpredictable due to cache's adaptive and dynamic behavior. Real-time applications are subject to operational deadlines and execution time predictability is considered mandatory to support them. Studies show that cache locking helps determine the worst case execution time (WCET) and cache-related preemption delay. In this chapter, we introduce an effective cache locking technique to enhance the predictability of an embedded system running real-time applications. We propose an algorithm that selects the blocks that may cause more cache misses in the same execution. We obtain hit ratio and mean delay for both cache analysis (no cache locking) and cache locking. Experimental results show that our proposed cache locking scheme improves predictability and performance up to locking 25% of the cache size, after that (if more blocks are locked) predictability may be further enhanced by sacrificing performance.

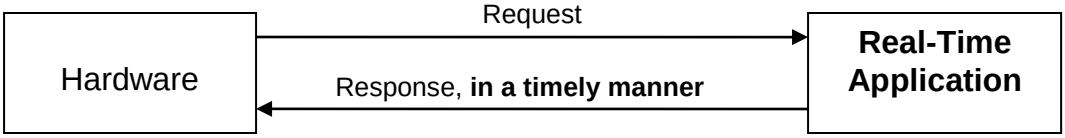
5.1 Introduction

Real-time multimedia applications running in embedded systems are becoming more popular day by day. Processing real-time applications on embedded systems is a

significant challenge for the memory subsystem. In a real-time system, the hardware and software are subject to operational deadlines from event to system response (a real-time constraint). A non-real-time system has no deadline, even if fast response is desired. Figures 5.1(a) and 5.1(b) show two systems – running non-real-time and real-time applications, respectively. In a real-time system, hardware requests are satisfied in a timely manner by the real-time operating system. A real-time deadline must be met, regardless of the system load.



(a) System running a non real-time application. Hardware requests are satisfied as soon the responses are ready.



(b) System running a real-time application. Hardware requests must be satisfied before the deadline.

Figure 5.1: Systems running non real-time and real-time applications.

Execution time predictability is a crucial factor for the success of real-time complex systems. The presence of cache is an important source of unpredictability due to its adaptive and dynamic characteristics; as a result, programs may behave in an unexpected way and it may be difficult to predict their execution time. Extensive research has been done to predict the worst-case behavior of embedded applications in order to determine the safe and precise bounds on tasks' worst-case execution time (WCET) and cache related preemption delay. In [140], the impact of data caches on predictability in

multitasking hard real-time systems is discussed. Cache locking mechanisms adapt caches to the needs of real-time systems. Recent studies show that cache locking reduces the time required to perform a memory access and improves predictability by removing both intra-task and inter-task interferences [108]. However, excessive cache locking decreases performance as effective cache size decreases. In this chapter, we focus on improving predictability of a real-time embedded system with little or no negative impact on performance/power ratio by developing an efficient cache locking scheme.

5.2 Cache Locking in Real-Time Single-Core Systems

Cache locking is a technique to hold a set of memory blocks inside the cache for the entire duration of the execution time [21], [22], [97]. Once a block is locked into the cache, the replacement algorithm excludes it to be removed until the application is completed. Cache locking has proven some potential to improve the execution time predictability of single-core systems running real-time applications. However, aggressive cache locking may decrease the performance by increasing cache misses (as the effective cache size decreases).

Currently available cache locking schemes are either lengthy or not-so-accurate. For example, [99] introduced two algorithms to address predictability – one for minimizing utilization and the other one for minimizing interferences. The genetic algorithm used in [25] does not guarantee selecting the right blocks to be locked. In this chapter, we introduce an efficient cache locking scheme that is based on the static tree-graph generated by the Heptane (Hades Embedded Processor Timing ANalyzEr) tool [162]. Heptane generates a tree-graph for C source files. The main objective of this

scheme is to lock the blocks that cause more misses if not locked. Figure 5.2 shows the major steps of our proposed cache locking scheme.

:: Major Stapes in our Proposed Cache Locking Scheme
Input: Instruction Block (IB) Address Miss info from Heptane tree-graph
Output: Instruction cache locking XML file
START:

- Read the Input File (created by off-line analysis)
- Create IB-Address Miss Block List
- Sort IB-Address Miss Block List (the block that causes the most misses becomes the number one candidate to be locked, and so on)
- List the Candidate Blocks (based on cache size, line size, and percent of locking)
- Create Instruction Cache Locking XML File

END

Figure 5.2: Major steps of proposed cache locking scheme.

The syntax tree is a tree whose nodes represent the structure of programs in the high-level language and whose leaves represent basic blocks. Leaves in the syntax tree coincide with the nodes in the control-flow graph. Appendix B shows Heptane generated tree-graph for FFT code. From the tree graph, in static analysis we collect the name of the node, number of instructions, the total number of cycles, and cache miss information for each node. From the off-line analysis we determine which code section of the source file causes more misses. We divide the analysis in several parts including the root node of the main C source file, the calling function for the C source file, all leaf node analysis for the root node and top loop node level analysis. We collect instruction block (IB) addresses' cache miss information based on the tree graph and we generate an instruction cache locking XML file (with the block addresses that should be locked). In order to implement

the static cache locking scheme, a small routine is required to be executed at the system start-up to load the content of the cache with the selected IB-address values and lock the cache blocks so that its contents remain available during the whole system execution.

This cache locking scheme helps determine the right amount of correct blocks to be locked which is the key to improve both the predictability and performance. This block selection algorithm may also be used for pre-fetching and pre-loading the cache.

5.3 Simulation

We simulate a real-time single-core system to evaluate the impact of our proposed cache locking scheme on the predictability and performance. Simulation details are presented in the following subsections.

5.3.1 Assumptions

Important assumptions for modeling the target architecture and running the simulation program include the following,

- A single-core architecture is considered and only I1 cache locking is implemented.
- Modified cache replacement strategy is used for I1. But random cache replacement strategy is considered for D1 and CL2.
- For both CL1 and CL2, write-back memory update policy is used.
- The average delay introduced by the bus that connects CL2 and the main memory is 10 times longer than the average delay introduced by the bus that connects CL1 and CL2.

5.3.2 Simulated Architecture

Heptane provided target architecture considered in this experiment is the simplified Pentium processor (P1 like the P54C) processor from Intel Corporation. The architecture model consists of a BTB (branch target buffer) and a cache system and a memory description. No data cache is modeled. Only one-level instruction cache is considered. One of the two integer pipelines is simulated. Furthermore, branch prediction module is kept disabled. In this study, we consider an instruction cache with cache size ranges from 2 KB to 32 KB, line size from 32 to 512 Bytes, and the associativity level from 1 (direct-mapped cache) to 16 (set associative cache). In this simulation, an instruction is assumed to execute in 1 clock cycle in the case of a cache hit, and 10 clock cycles otherwise.

5.3.3 Simulation Tools and Workload

Heptane package is used to develop the model and to run the simulation program in this work. Three applications, Fast Fourier Transform (FFT), Matrix Inversion (MI), and Discrete Fourier Transform (DFT), are used to run the simulation. The actual "C code" of these applications is used as the input to the simulation. Table 5.1 shows the code size, number of instructions, and computing time of these applications.

Table 5.1: Some statistics of FFT, MI, and DFT applications

Applications	Code Size (KB)	Number of Instructions	Computing Time (no locking) (Kilo Cycles)
FFT	2.34	365,184	121,235
MI	1.47	227,518	186,519
DFT	1.16	171,307	258,456

5.3.4 Experimental Results

In this work, we implement our proposed cache locking algorithm to obtain the hit ratio and mean delay for FFT, MI, and DFT applications in a single-core system. Using Heptane, we obtain the computing time and WCET.

Hit ratio and mean delay obtained for 2 KB and 4 KB cache sizes by varying the amount of cache locked, from 0% to 50% (with 12.5% increment) of the cache size, using FFT code is shown in Table 5.2.

Table 5.2: Cache locking and hit ratio for FFT application

LINE SIZE 128 BYTES, ASSOCIATIVITY 8-WAY				
Cache Locking (%)	Cache Size 2K		Cache Size 4K	
	Num of Block Locked	Hit Ratio (%)	Num of Block Locked	Hit Ratio (%)
0.0	0	95.00	0	100.00
12.5	3	95.71	6	100.00
25.0	8	96.43	16	100.00
37.5	11	96.12	19	100.00
50.0	16	95.27	19	100.00

Experimental results indicate that the hit ratio is the maximum (i.e., performance is also the maximum) at 25.0% cache locking for FFT when cache size is 2KB. However, cache locking has no positive impact on hit ratio for 4KB cache as FFT code entirely fits into the cache.

We conduct theoretical and mathematical analysis of the impact of the cache locking on hit ratio and present those results along with simulation results using FFT code. Assuming 80% hit ratio (without cache locking), 50% cache size locking produces 100% hit ratio when B = C / 2, where B is the number of blocks that causes misses and C is the total number of blocks in the cache. Also, as illustrated in Figure 5.3, for B = C and B = 2 * C, hit ratio increases gradually but not as sharply as does for B = C / 2. However, the simulation results show that initially hit ratio increases when cache locking is applied. But beyond 25% locking, hit ratio decreases. This is because in theoretical study it was not considered that as locked cache size increases, the effective cache decreases which cause cache hit rate to decrease.

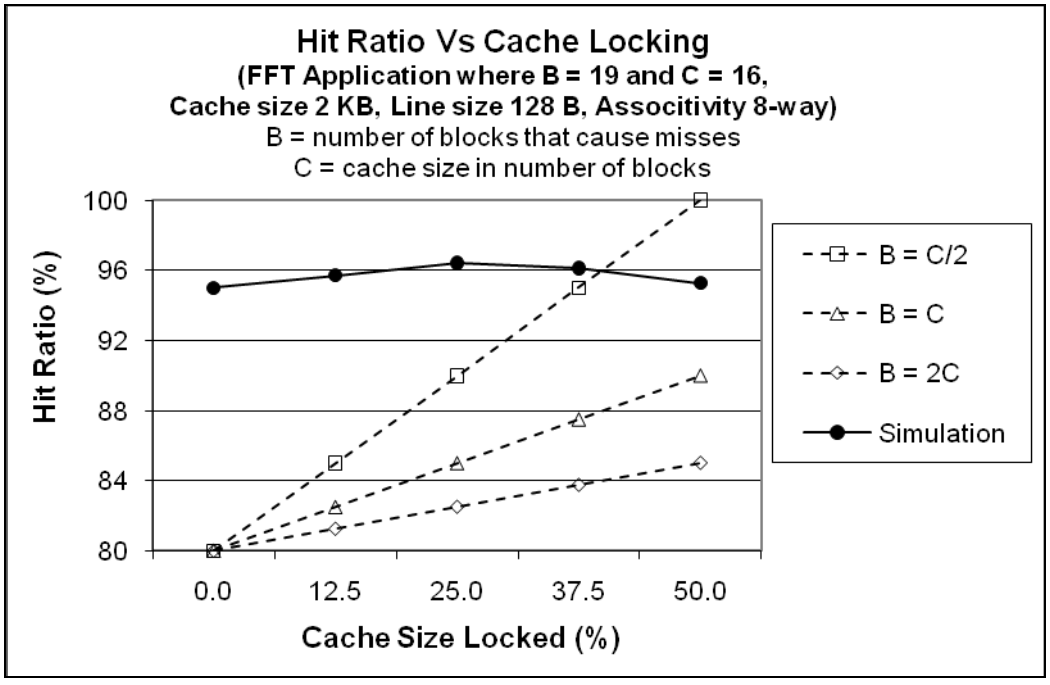


Figure 5.3: Hit ratio versus cache size locked.

We obtain the mean delay for FFT, MI, and DFT by varying the cache locking capacity. As shown in Figure 5.4, FFT causes the maximum delay followed by MI and DFT. For FFT, we notice that the minimum mean delay (i.e., maximum performance) is at 25% cache locking. But for MI and DFT, mean delay remains unchanged. This is because MI or DFT code size is smaller and that fits entirely in 2 KB I1 (but FFT code does not).

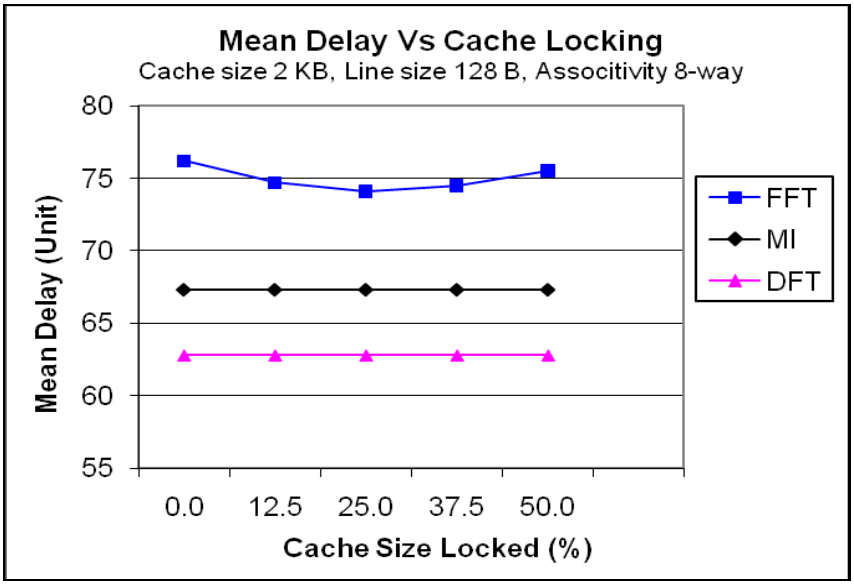


Figure 5.4: Mean delay versus cache size locked.

In our experiment, if cache locking capacity is increased beyond 25%, predictability increases but performance decreases (i.e., mean delay increases). So, for maximum performance we chose the cache locking capacity at 25% of the cache size. In the following subsections, we discuss the impacts of cache size, line size, and associativity level on the performance of FFT, MI, and DFT applications at 25% cache locking.

We investigate the impact of cache size on mean delay (i.e., performance) for FFT, MI, and DFT applications. We keep line size fixed at 128 Bytes and associativity level at 8-way. Experimental results are shown in Figure 5.5. For the given line size and associativity level, the mean delay of both static cache analysis (no cache locking) and cache locking decreases (i.e., performance increases) with the increase in the cache size for FFT when cache size is increased from 2 KB to 4 KB. No changes in mean delay for FFT for cache size bigger that 4 KB. There is no change in mean delay for MI and DFT due to the increase in cache size.

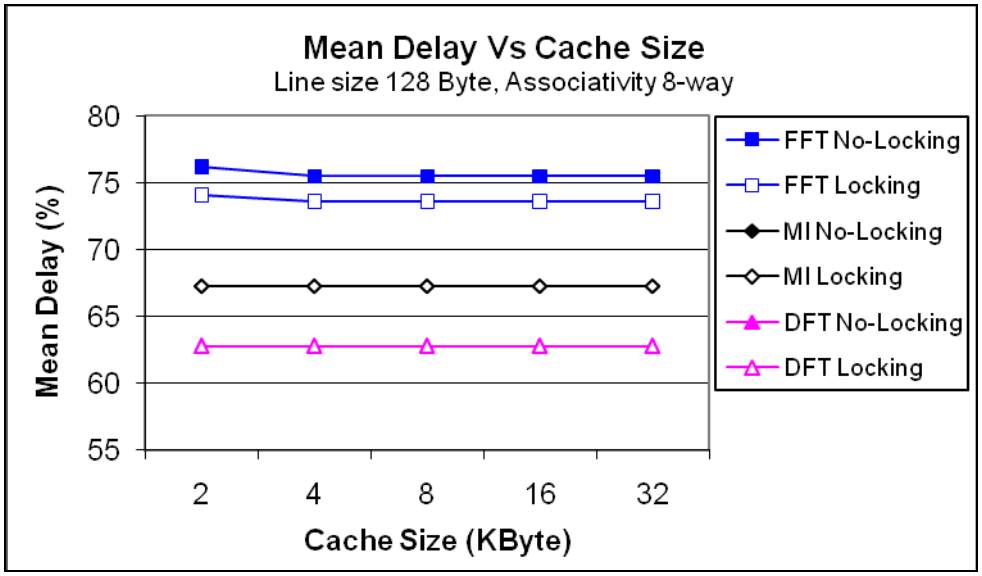


Figure 5.5: Mean delay versus cache sizes.

Mean delay obtained for both static cache analysis (no cache locking) and cache locking for FFT, MI, and DFT applications are shown in Figure 5.6 with varying line sizes. For the cache size fixed at 2 KB and the associativity level fixed at 8-way, the mean delay decreases (i.e., performance increases) with increases in the line size from 32 to 128 Bytes for FFT; for line sizes higher than 128Bytes, mean delay increases (i.e., performance decreases). No change is noticed in mean delay for MI and DFT due to the increase in line size.

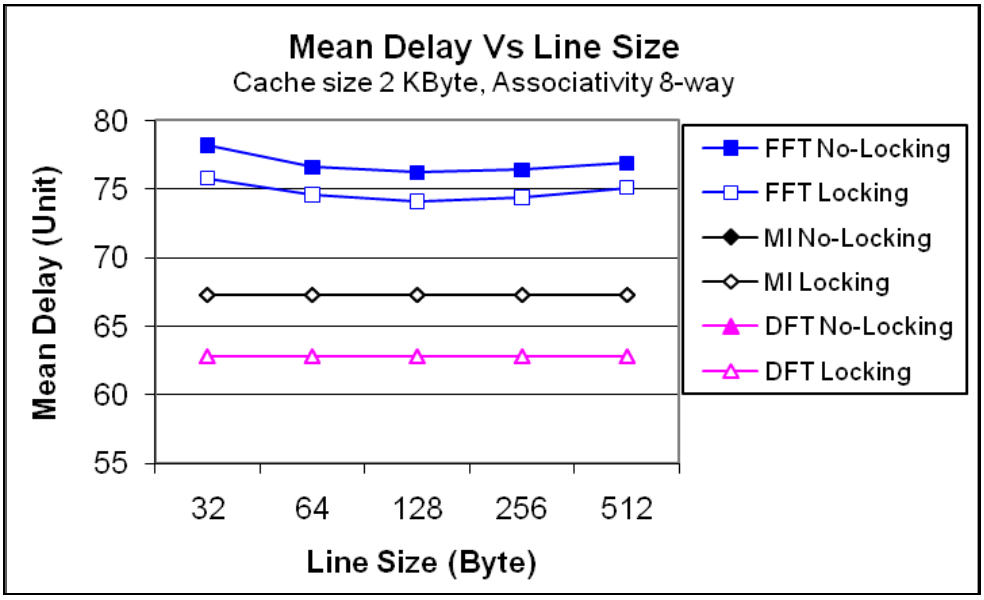


Figure 5.6: Mean delay versus line sizes.

Finally, we obtain the mean delay for both static cache analysis (no cache locking) and cache locking by varying associativity level as shown in Figure 5.7. Results show that for a cache size of 2 KB and a line size of 128 Bytes, the performance of static cache locking scales better than the one of no cache locking with an increasing level of associativity for FFT. Cache locking benefits from the increasing associativity level in eliminating both intra-task and inter-task interference. Again, there is no change in mean delay for MI and DFT due to the changes in associativity level.

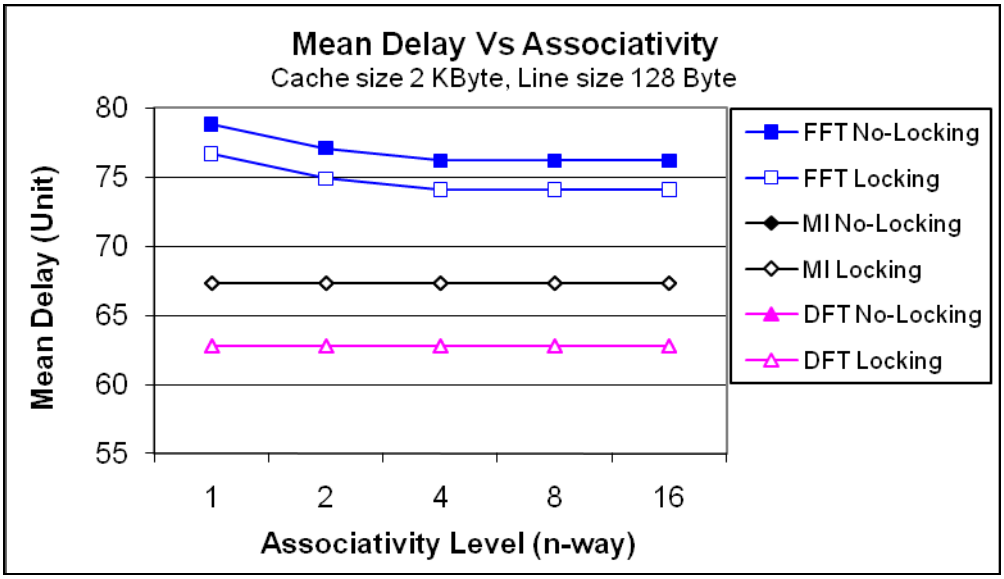


Figure 5.7: Mean delay versus levels of associativity.

5.4 Summary

The demand for real-time applications in embedded systems is growing. Execution time predictability is an important design factor for any real-time system. Cache improves performance but introduces challenges to improving execution time predictability [54]. It has been shown that for embedded systems with a well known

workload, static cache locking helps to determine the worst case execution time and cache-related preemption delay. In this chapter, we introduce an effective cache locking scheme that makes the real-time embedded system more predictable. We use the FFT, MI, and DFT application codes as inputs to our simulation program developed by Heptane package [162]. We obtain hit ratio and mean delay for both static cache analysis (no cache-locking) and cache locking. Experimental results show that our cache locking algorithm improves predictability when application does not fit entirely in the cache. Also, performance can be improved when the right cache blocks are locked and appropriate cache parameters are used. Experimental results also show that predictability can be further enhanced by sacrificing performance.

CHAPTER 6

MISS TABLE BASED CACHE LOCKING WITH VICTIM CACHE TO IMPROVE PREDICTABILITY AND PERFORMANCE/POWER RATIO

Cache memory increases execution time unpredictability and makes it difficult to support real-time applications. Execution time unpredictability becomes worse in multi-core processors due to the presence of multi-level caches. Multi-level caches consume significant amount of energy as caches are power hungry. Modern embedded systems require tremendous amount of processing speed (i.e., high-performance) to support real-time complex applications. Studies indicate that proper implementation of cache locking may improve the predictability and performance/power ratio. In this chapter, we propose a Miss Table at cache level to improve cache locking performance and we use victim cache between CL1 and CL2 to increase cache hit ratio. This scheme is effective for both single-core and multi-core embedded systems. We consider a multi-core architecture with shared CL2, where each core has a VC between its CL1 and the shared CL2. Cache parameters are first optimized for optimal performance/power ratio. Then the blocks that are expected to cause more misses are locked (up to a certain number of blocks using MT) for the maximum predictability. It is observed that the proposed scheme with MT and VCs significantly improves predictability, reduces mean delay, and reduces total

power consumption for smaller amount of locked cache (up to 25% of the cache size). It is also observed that predictability can be improved even further by locking more blocks at the expense of performance/power ratio.

6.1 Introduction

Single-core processors are simple (with one processing core and its cache memory organization) and cheap compared to multi-core processors. That's why single-core processor is the first choice for some embedded systems. However, embedded systems are adopting multi-core processors in their architectures to meet the requirement for high performance/power ratio. The popularity and demand of multi-core real-time systems are increasing in both desktop and embedded markets [9], [127]. Unlike single-core architecture, multi-core architecture is complicated as it has multiple cores and multiple (levels of) caches. In a multi-core processor, two or more independent cores are combined into a die. In most cases, each processor has its own level-1 cache, split into instruction and data caches. The multi-core processors usually have other-levels of caches (CL2, CL3, etc) [47], [102], [109], [166]. CL1 is usually private to each core, but CL2 (and higher level caches) can be configured in a number of ways. For example, CL2 may be private to each core or shared by all cores. Therefore, cache optimization becomes very important (at the same time, very difficult) in a multi-core system with multi-level caches. Studies show that cache parameters significantly influence system performance [5], [10], [33], [35]. Multi-core architectures are more suitable for real-time applications, because concurrent execution of tasks on a single processor, in many respects including energy requirement, is inadequate for achieving the required level of performance and

reliability. The problem is that cache introduces execution time unpredictability and real-time applications demand execution time predictability and cannot afford to miss deadlines. Therefore, it becomes a great challenge to support real-time applications on multi-core systems.

Studies show that for single-core systems, cache locking improves the predictability [11], [12], [24], [97], [139]. Cache locking is the ability to prevent some or all of the instruction or data cache blocks from being overwritten. Cache entries can be locked for either an entire cache or for individual ways within the cache. Entire cache locking is inefficient if the number of instructions or the size of data to be locked is small compared to the cache size. In way locking, only a portion of the cache is locked by locking ways within the cache. Unlocked ways of the cache behave normally. Way locking is an alternative of entire locking. Using way locking, Intel Xeon processor may achieve the performance of using local memory without cache by Synergistic Processing Elements (SPEs) in IBM Cell processor architecture [18], [19], [43], [117], [124], [157].

The integration of billions of transistors in a single chip is now possible. As a result, the multi-core design trend is expected to grow for the next decade. To the best of our understanding, current multi-core processors are not able to take full advantage of cache locking, because most existing cache locking mechanisms are developed for single-core processors. In this chapter, we introduce Miss Table based cache locking scheme with victim cache, which is suitable for both single-core and multi-core systems, to improve the predictability and performance/power ratio by selecting the blocks wisely for cache locking and replacement and by storing victim blocks from level-1 caches and supporting stream buffering.

6.2 Improving Predictability and Performance/Power Ratio

Execution time predictability is one of the key design factors for any real-time system. Also, power consumption is very crucial for embedded systems. As a result, increasing performance and predictability and decreasing power consumption is the primary focus of cache optimization in real-time embedded systems. Multi-core architectures are more complex than single-core architectures and caches in multi-core make the unpredictability even worse.

Cache optimization techniques are proven to increase performance and reduce power consumption in embedded systems. Studies show that predictability in a single-core system can be improved by applying cache locking technique. In this chapter, we introduce a Miss Table based cache locking scheme with victim cache(s) in order to increase the predictability and performance/power ratio for both single-core and multi-core real-time embedded systems.

6.3 Introducing Miss Table at Cache Level

Cache locking is a known technique to hold a set of memory blocks inside the cache for the entire duration of the execution time [24], [97]. Once a block is locked into the cache, the replacement algorithm excludes it from being removed until the execution is completed. Cache locking improves predictability. However, foolish block selection and aggressive cache locking may decrease the performance by increasing cache misses (as the effective cache size decreases). Studies show that performance increases with the increase of the cache locking capacity for smaller values (0% to 25% of the cache size).

We introduce a table that contains information about all or some important blocks those cause more cache misses related to the code being processed. We call it a Miss Table (also referred as MT). MT is, primarily, introduced for improved cache locking performance. MT also helps improve cache replacement performance. MT is implemented at cache level in such a way that it can be accessed from the cache(s) where cache locking is implemented and all other cache(s) below that level. For level-2 cache locking, MT should be accessed by CL2, CL1, and victim caches (if any). Block addresses are sorted in descending order of the number of misses they cause. For each application/function, after post-processing the tree-graph generated by Heptane package, block address information is prepared for the MT. During system initialization, MT is populated with the corresponding block addresses. MT information is used to select the blocks to be locked or replaced. For better performance, MT should store information about at least 'N' cache blocks, where N = total cache size / size of a cache block.

6.3.1 Miss Table Workflow

The schematic diagram of an embedded system with one core, Miss Table, and cache memory subsystem is shown in Figure 6.1. MT is loaded during the initialization process with block information selected from off-line analysis. Cache blocks are locked (inside CL1 or CL2) using the information stored in MT. During execution, when a new block is brought into CL2 (or CL1), it is checked using the MT information if the block should be locked in case of CL2 locking (or CL1 locking). 'L' bit is set to indicate a block is locked. If the requested memory block is not found in CL1 (even though CL1 is full), CL2 is checked for the requested block. In this case, a victim block (VB) in CL1 is

selected using the Miss Table. A victim block should be a block with ‘L’ bit is clear (i.e., block is not locked) and minimum number of misses. Finally, a CL1 victim block is replaced by the requested memory block (from CL2).

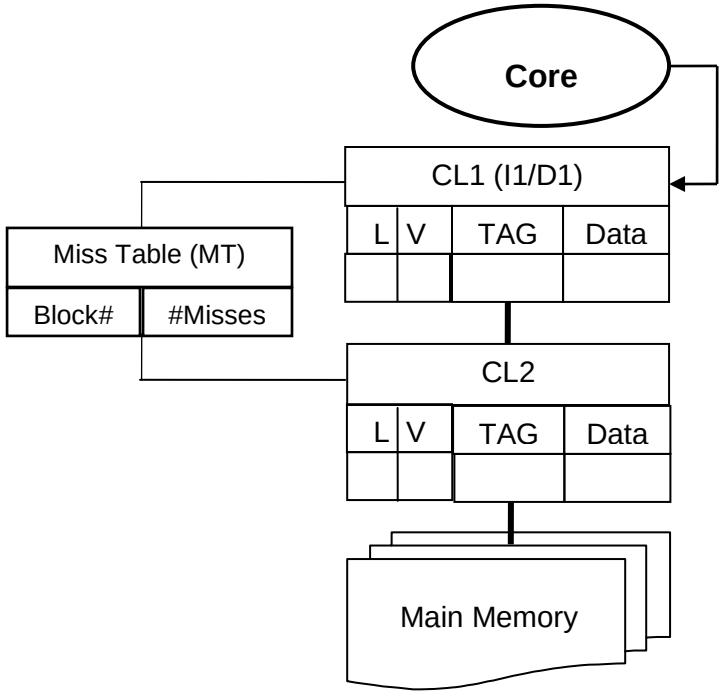


Figure 6.1: Schematic diagram of an architecture showing one core, Miss Table, and cache memory hierarchy.

6.3.2 Victim Block Selection Criteria

Using the Miss Table information, a modified replacement policy is used to select the victim block from CL1. In case of a CL1 miss, a victim block should be selected. This policy selects a block that is not locked and has the minimum number of misses. In case of a tie in the number of misses, a block should be selected randomly. Summarized below are the important criteria of a victim block:

a) A victim block cannot be a locked block.

b) A victim block should have the minimum number of misses among the un-locked blocks (in CL1) at the time of selection.

c) In case of a tie in the number of misses, a block should be selected randomly.

6.4 Victim Cache between Level-1 and Level-2 Caches

Victim cache (also referred as VC) is considered as a bypass cache and a predictor table between CL1 and CL2. Victim blocks from CL1 are temporarily stored in VC (instead of destroying them). This method of using VC reduces access latencies by determining whether a load should bypass the main cache hierarchy and issue an early load to main memory [75]. The victim cache hierarchy is suitable for systems with limited cache-memory area (like embedded systems) and applications that perform a large amount of memory accesses (like multimedia) [155]. Victim cache hierarchy has the potential to reduce execution time and improves predictability for some applications. Also, victim cache hierarchy offers improved cache energy consumption with comparable performance gain by reducing cache misses [4].

Figure 6.2 shows the schematic diagram of an architecture with a core, MT, VC, and cache memory hierarchy. VC resides between CL1 and CL2 and can access MT. Requested memory block is first checked into CL1. If not found in CL1, VC is checked; if not found in VC, CL2 is checked. Finally, if not found in CL2, pre-fetching or stream buffering (an improved version of pre-fetch when multiple blocks are fetched from main memory) is performed. Requested block goes to CL1 (from main memory via CL2). In case of stream buffering, the additional blocks go to VC. CL1 and VC select victim block using the Miss Table. In this work, cache locking is implemented in CL2.

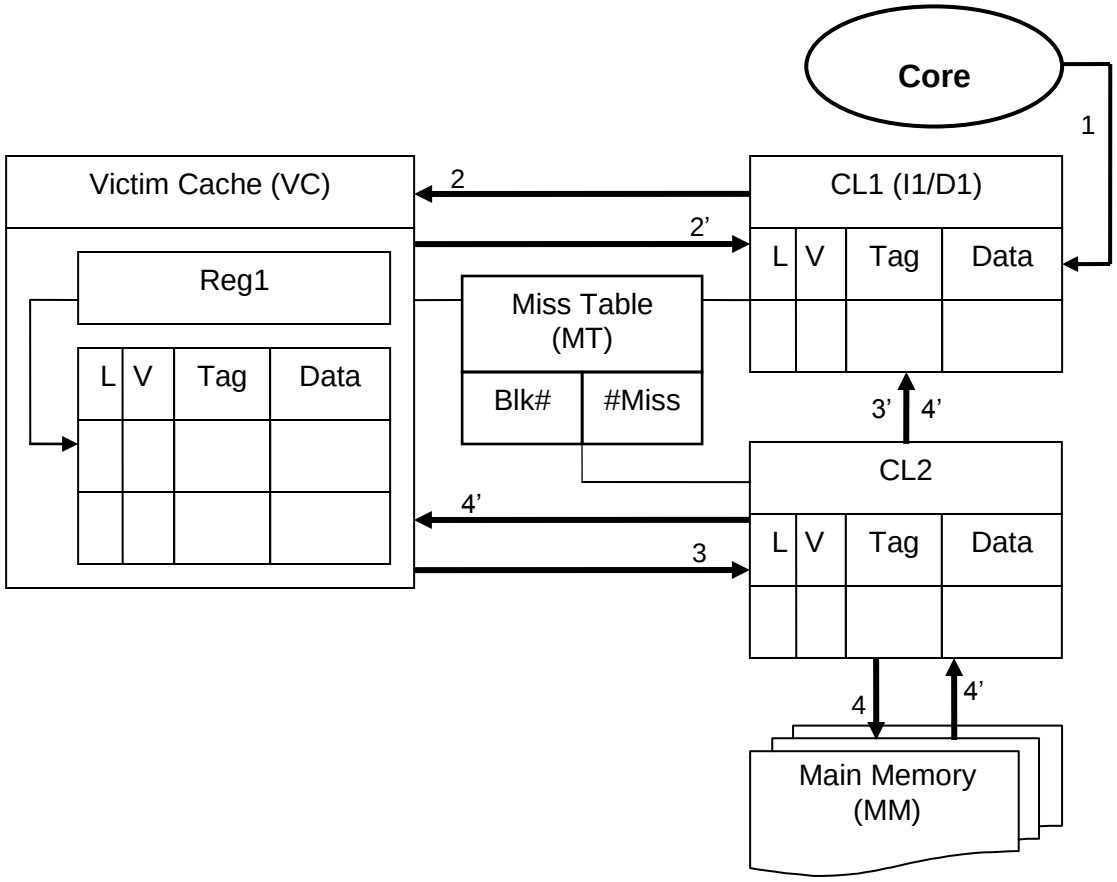


Figure 6.2: Schematic diagram of an architecture showing one core, MT, victim cache, and cache memory hierarchy.

6.4.1 Flow inside the Core

Flow diagram inside a core with the proposed Miss Table and victim cache is shown in Figure 6.3. In case of a CL1 (I1 or D1) miss, MT is used to find the victim block (VB). If a CL1 miss is followed by a VC miss and VC is full, then a non-dirty VC block (or multiple VC blocks for stream buffering) is selected using MT information. If the request is not satisfied from CL2, stream buffering (updated pre-fetch) is performed. In case of stream buffering, the requested block goes to CL1 (as expected) and the additional blocks go to the VC.

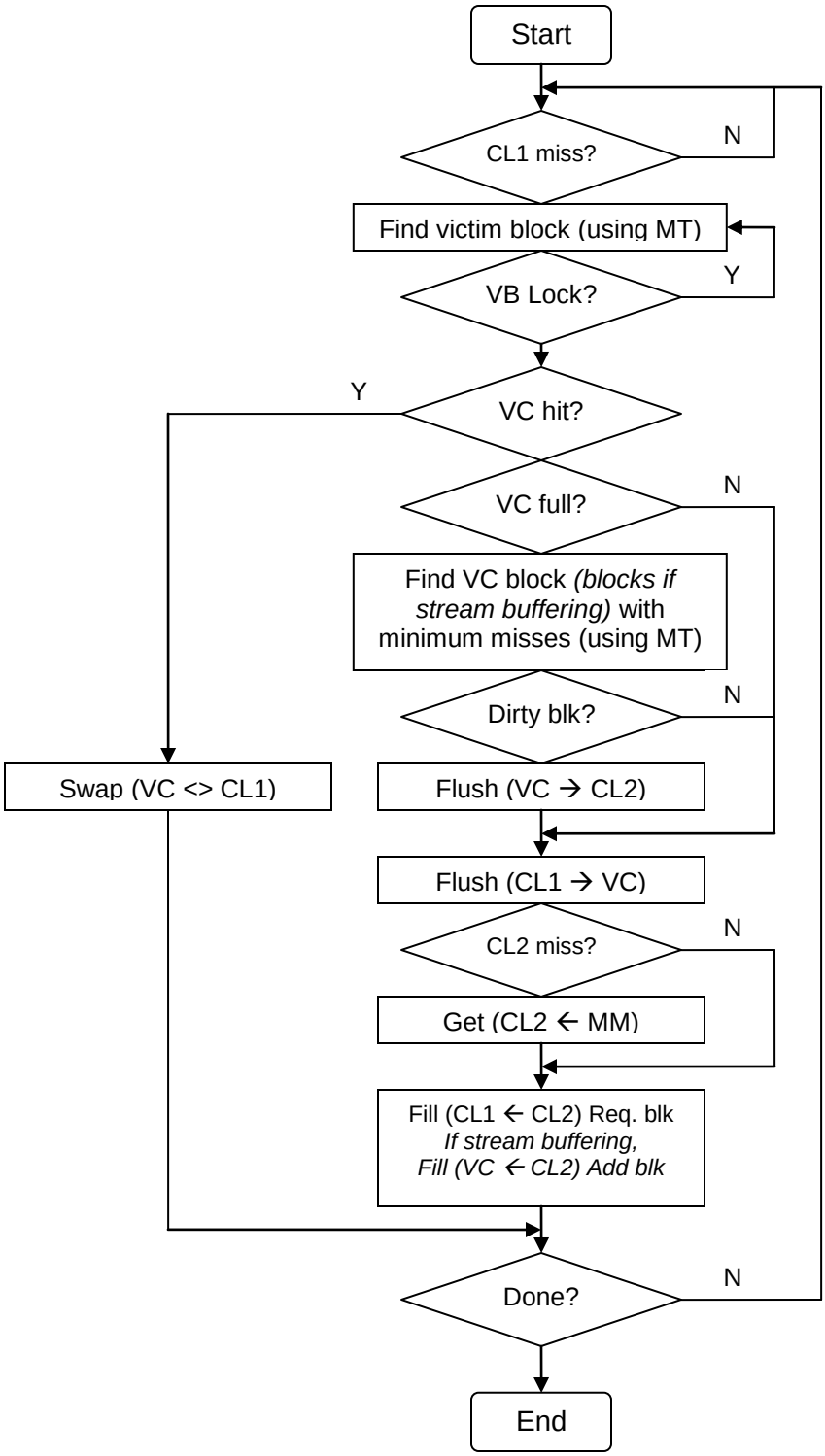


Figure 6.3: Schematic diagram of the flow inside the architecture with one core, MT, VC, and cache memory hierarchy.

6.4.2 Control Logic for the Victim Cache

The main purpose of the victim cache is to hold the victim blocks from CL1. In addition, it can be used when stream buffering is performed. Figure 6.4 shows the control logic for VC. It is important to note that VC can be turned off or on functionally. If VC is functionally off, in case of a CL1 miss, CL2 is checked (bypassing the VC) for the requested block. When VC is being used, in case of a VC hit, a swap between CL1 block and VC block is done using a register (Reg1 in Figure 6.2). If stream buffering is used, then the additional blocks fetched from main memory is loaded into VC (the requested block goes to CL1).

```
:: Victim Cache Control Logic
START:
If (CL1_Request) Then
  If (VC_Disable) Then
    Done // Perform normal as if VC does not exist
  Else // VC_Enable
    Reg1 ← CL1 cache line victim block (VB)
    If (VC_Hit) Then
      CL1 ← VC cache line (Hit)
      VC ← Reg1
    Else // VC_Miss
      If (VC_Full and Dirty_Block) Then
        CL2 ← VC cache line(s) with minimum misses
      End If
      If CL2 miss, Issue Pre-fetch or Stream_Buffering
      CL1 ← Reference block from CL2 or MM
      IF (Stream_Buffering) Then
        VC ← Additional blocks from CL2 or MM
      End If
    End If // VC_Hit / VC_Miss
  End If // VC_ Disable / VC_Enable
End If // CL1_Request
END
```

Figure 6.4: Control logic for the proposed victim cache and stream buffer.

6.5 Additional Techniques

Additional techniques, which have been proven to improve the predictability and performance/power ratio by increasing hit ratio include selective pre-loading, pre-fetching, and stream buffering.

Selective pre-loading: Selective pre-loading is a technique to load the caches with previously selected blocks to reduce compulsory (cold start/first reference) cache misses. Studies show that if blocks are selected wisely then selective pre-loading improves both performance/power ratio and execution time predictability by reducing compulsory misses.

Pre-fetching: Pre-fetching is a technique that allows memory subsystem to import data into the cache before the processor needs it. Pre-fetching may improve predictability for (hard) real-time systems. However, aggressive pre-fetching can lead to cache pollution and also increase memory traffic [94]. In distributed shared memory (DSM) systems, remote memory accesses take much longer than local ones, and hence data pre-fetching should be effective for such systems [64].

Stream buffering: Stream buffering is an improved pre-fetching technique where multiple blocks (instead of one) are brought from main memory when a pre-fetch request is executed. Studies show that stream buffering is a very powerful technique to improve performance and predictability and to reduce total power consumption [44], [58].

In this chapter, we simulate a multi-core architecture where we implement selective pre-loading using MT and stream buffering using victim caches.

6.6 Workload Characterization for Cache Locking

We find currently available workload and workload characterization methodology inadequate to simulate cache locking in multi-core architecture. Therefore, we propose a workload characterization methodology that can be used to simulate cache locking in both single-core and multi-core systems. Proposed workload characterization technique has the following three phases – code division, code estimation, and block selection.

6.6.1 Phase-I: Code Division

In phase-I, we analyze the application(s) and divide the code into smaller segments if needed (see Figure 6.5). In order to support small applications, the applications are mapped directly among the cores. Each application should be assigned to a free core (entire application is considered). In order to support a large application, the code is divided into smaller (end-to-end) functions in such a way that a function can be assigned to a core (each function is considered separately). Code division is crucial for big applications (in terms of code-size).

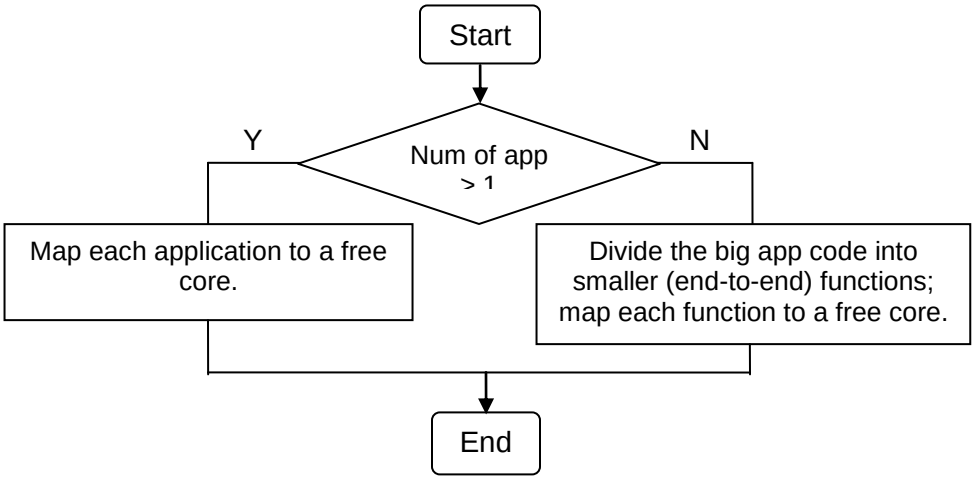


Figure 6.5: Code division workflow diagram.

In this work, we consider real-time MPEG4, H.264/AVC, FFT, MI, and DFT applications of which MPEG4 and H.264/AVC code is divided into smaller segments. After Phase-I is completed, phase-II and phase-III can be done simultaneously.

6.6.2 Phase-II: Code Estimation

In phase II, we estimate important operations for each application (in case of small applications) or function (in case of large applications). Major steps in phase II is shown in Figure 6.6. Code estimation can be done manually for smaller code segments.

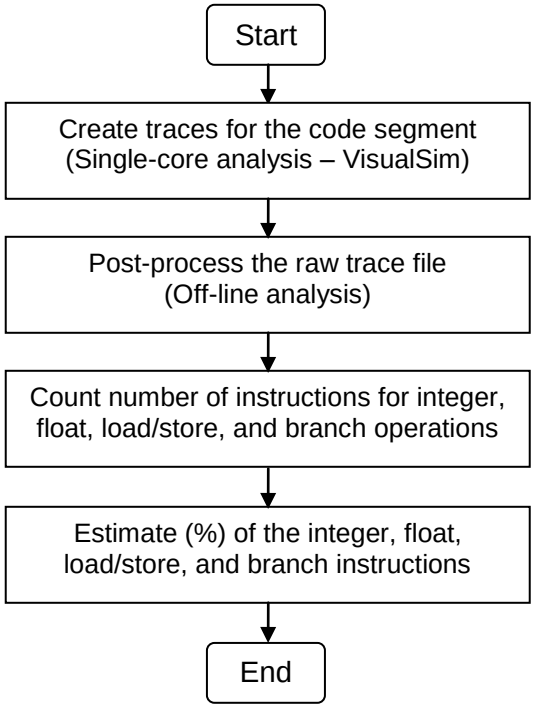


Figure 6.6: Code estimation workflow diagram.

The type and percent value of the operations are used to obtain the mean delay per task and total power consumption.

In this work, VisualSim code annotation technique is used to obtain the number of integer, floating-point, load/store, and branch operations. Table 6.1 shows the estimated value for FFT algorithm obtained from Phase-II. Complete FFT code used in this work is given in Appendix A.

Table 6.1: Code estimation for FFT

Type of Operation	Number of Operation (%)
Integer	18
Floating-point	71
Load/Store	9
Branch	2

6.6.3 Phase-III: Block Selection

Finally in phase III, we select the blocks for cache locking. Major steps in this process are shown in Figure 6.7. A tree-graph for each code segment is created using Heptane (Hades Embedded Processor Timing ANalyzEr) tool. The tree-graph for FFT code is shown in Appendix B. Contextual information such as block address and cache miss are stored in the leaves (example: CALL, CODE) and control-flow nodes (example: SEQ, LOOP) (see Figure A in Appendix B). The leaves represent the basic blocks and the nodes of a tree-graph represent the structure of a program in the high-level language. From the tree graph, we collect the number of instructions and cache miss information for each node. From the off-line analysis we determine which blocks make more misses. By post-processing the information collected from the tree-graph, we obtain the block address that should be locked.

The number of instructions and cache misses for sequence nodes (i.e., SEQ) and the number of instructions and cache misses for loop nodes (i.e., LOOP) are excluded in this work because we consider all code nodes (where a sequence node or a loop node represents a set of code nodes).

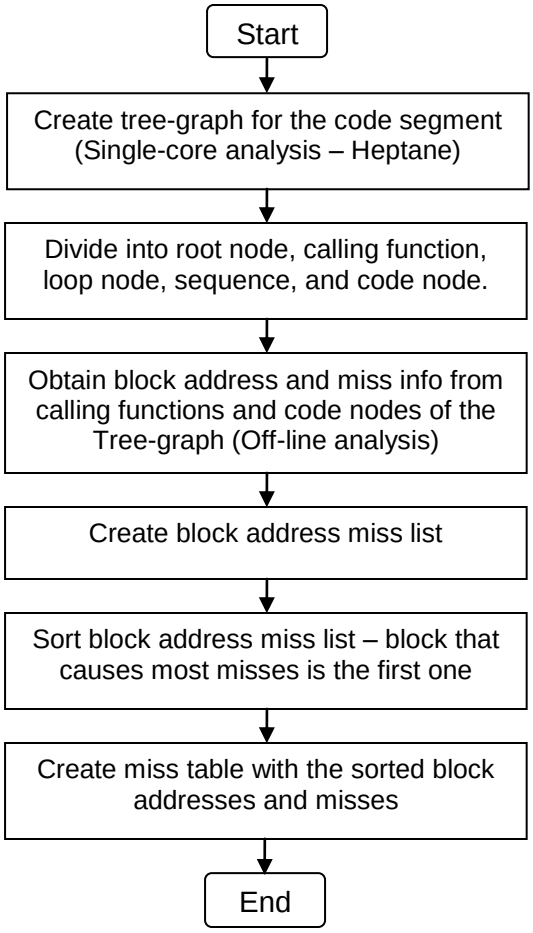


Figure 6.7: Block selection (for cache locking) workflow diagram.

Table 6.2 shows the block addresses and total misses (sorted in descending order of the number of misses) obtained for FFT algorithm. Blocks are selected for locking depending on the cache size, line size, and locked cache size. For an example: if cache size is 2 KB, line size is 128 B, and associativity level 8-way, then number of blocks is 2*1024/128 = 16. So, MT should store information about top 16 blocks (or more) from Table 6.2. Now, locking 2 of 8 ways (i.e., 25% cache size) means 4 (25% of 16 is 4) blocks should be selected for locking. The best case scenario is that the first 4 blocks are selected to be locked.

Table 6.2: Sorted block address of FFT for cache locking

Cache size 2 KB, line size 128 B, locking 2 of 8 ways (25%)				
Num	**Block Address (Hex)**	**Total Cache Misses**	**Block in MT? (Yes/No)**	**Block Locked? (Yes/No)**
1	**0**	**35**	**Yes**	**Yes**
2	**80**	**33**	**Yes**	**Yes**
3	**300**	**32**	**Yes**	**Yes**
4	**100**	**28**	**Yes**	**Yes**
5	400	17	Yes	No
6	320	15	Yes	No
7	380	14	Yes	No
8	180	12	Yes	No
9	280	11	Yes	No
10	360	11	Yes	No
11	420	10	Yes	No
12	200	9	Yes	No
13	140	4	Yes	No
14	480	4	Yes	No
15	160	3	Yes	No
16	220	3	Yes	No
17	440	3	No	No
18	120	1	No	No
19	260	1	No	No

From Table 6.2, total cache misses is 246. By locking the first four cache blocks (for 25% locking), 124 cache misses can be avoided (i.e., cache miss is reduced by more than 50%). At system start-up, a small routine is required to load the content of the cache for the selected blocks and lock the cache during the whole execution time.

6.7 Modeling and Simulation

In the previous chapters, we have seen the positive impact of cache optimization and cache locking on performance, power consumption, and predictability. In Chapter 3, we model and optimize a single-core system to improve the performance. In Chapter 4, we optimize a multi-core system to improve performance/power ratio. In Chapter 5, we develop a promising cache locking scheme in a single-core system to improve the predictability. In this chapter, we model and simulate an Intel quad-core like multi-core architecture to explore the impact of Miss Table and victim caches on the predictability and performance/power ratio. We introduce a Miss Table that helps cache locking and block selection for replacement. The cache locking scheme is expected to improve predictability by locking more important cache blocks into CL2 (we simulate level-2 cache locking). According to this scheme, the blocks that are anticipated to cause more misses should be locked. Therefore, this cache locking scheme is expected to improve performance/power ratio by decreasing cache misses. We, also, use victim caches between CL1 and CL2. Victim caches are used to temporarily store the victim blocks from CL1 and store additional blocks fetched from main memory if stream buffering is used. Even though we are simulating a multi-core architecture in this work, this scheme can be used in both single-core and multi-core systems.

6.7.1 Assumptions

We make the following assumptions while simulating the proposed Miss Table based cache locking scheme with victim caches in this chapter.

- We simulate a multi-core system with 4 cores; each core has its own CL1 (split into I1 and D1) and CL2 is shared by all cores. It may be noted that we implemented cache locking in a single-core system in the previous chapter.
- Level-2 cache locking is considered in this work.
- MT is at cache level and can be accessed from CL1, CL2, and VCs (if any).
- Selective pre-loading, victim cache, and stream buffering are considered.
- Write-back memory update policy is used.
- Modified cache replacement strategy is used in CL1 and VCs (if any) to select blocks with minimum misses and in CL2 (to exclude locked blocks).
- The delay introduced by the bus that connects CL2 and the main memory (Bus2 in Figure 6.8) is 10 times longer than the delay introduced by the bus that connects CL1 and CL2 (Bus1 in Figure 6.8).

6.7.2 Simulated Architecture

We simulate an Intel quad-core like architecture, where each core has its own private CL1. CL1 is split into I1 and D1 for improved performance. A schematic diagram of the simulated multi-core system is shown in Figure 6.8. The architecture consists of one shared CL2. Two cores are connected to the CL2 using the same bus to reduce the bus contention. We introduce MT to improve cache locking performance and use VCs to improve performance/power ratio. Cache locking is implemented in CL2.

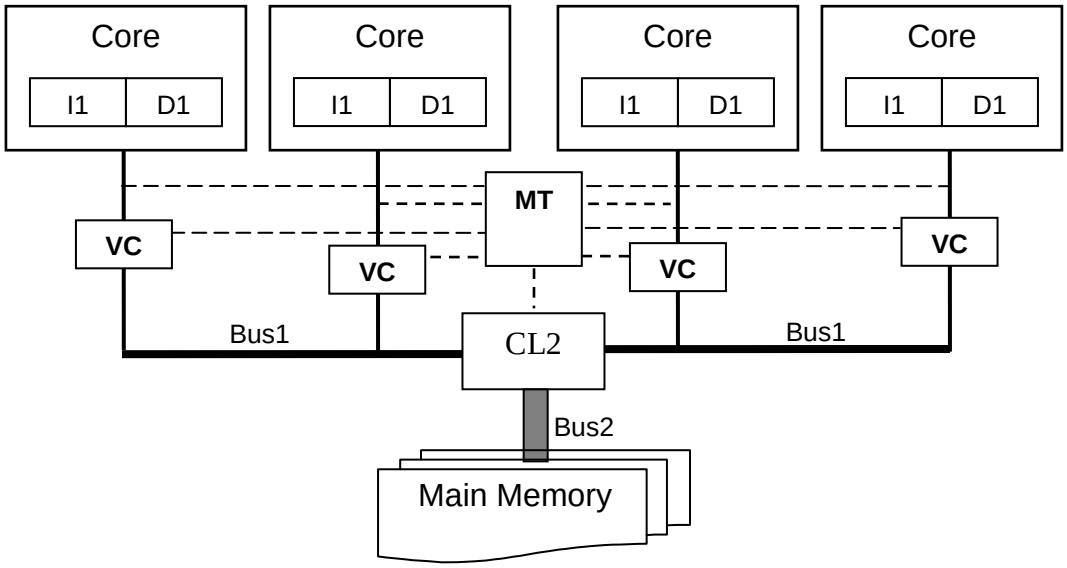


Figure 6.8: Simulated multi-core architecture with MT and VC.

MT contains information about cache blocks (obtained by off-line analysis) that cause misses (in descending order with respect to number of misses). MT is used to select a victim block from CL1. VCs are used to store victim blocks from CL1s. Also, VCs are used for stream buffering. The proposed MT based cache locking with VCs is expected to improve the predictability and performance/power ratio.

6.7.3 Major Steps in the Workflow

Major steps involved in the workflow of our proposed Miss Table based cache locking scheme with victim caches are shown in Figures 6.9a and 6.9b. As shown in Figure 6.9a, Miss Table is loaded with block address and cache miss information during the initialization so that (level-2) cache locking may be applied as needed. The cache misses with and without 25% cache locking are obtained using Heptane and (those cache misses are) used in VisualSim simulation program to obtain the mean delay per task and

total power consumption with and without 25% cache locking. The scheduler generates and assigns N (or less) tasks to N homogeneous cores until the total number of tasks are completed. Total number of tasks is equal to the total number of instructions in the code.

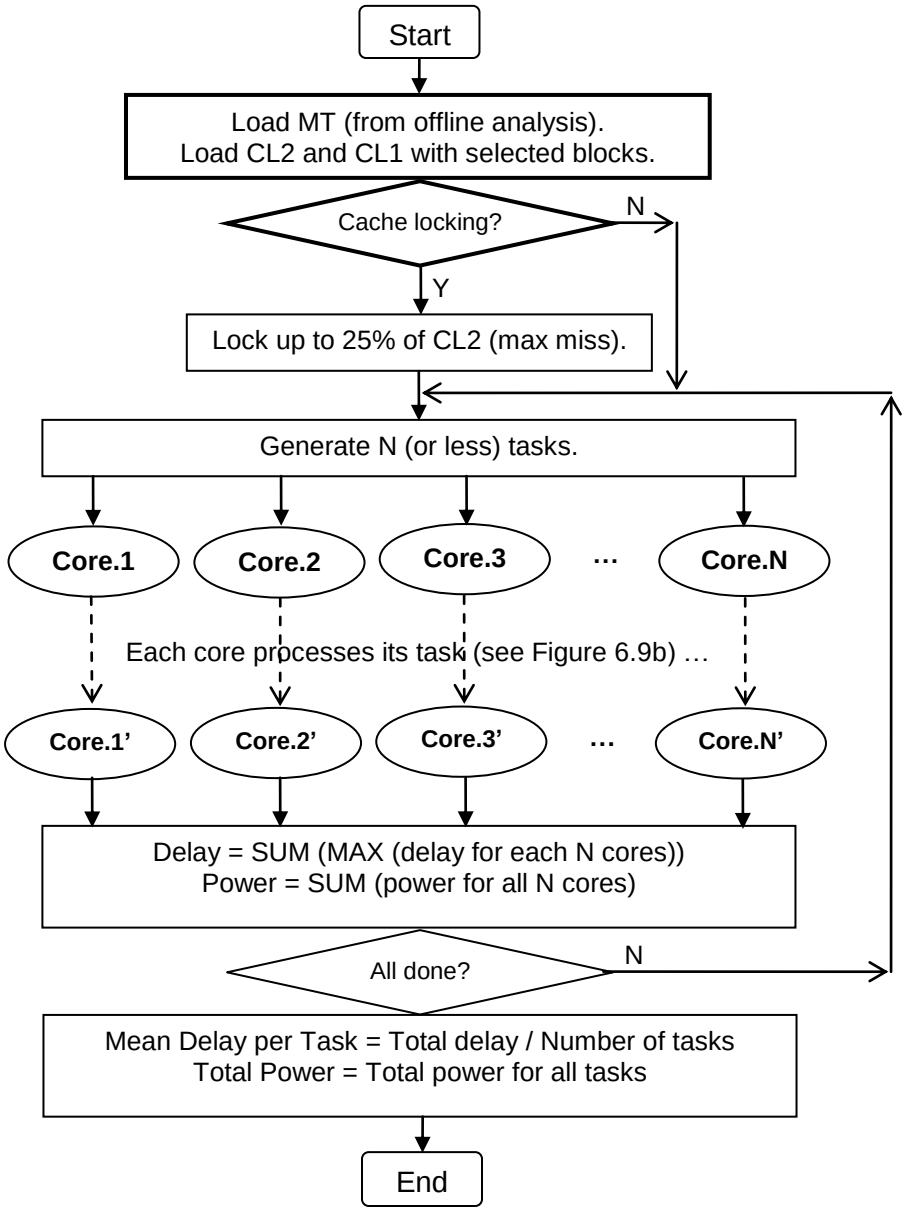


Figure 6.9a: Workflow diagram of the proposed cache locking scheme using Miss Table in an N-core system.

For each group of N tasks, the maximum delay is considered to obtain the mean delay per task and power consumed by all cores is considered to obtain the total power consumption. Figure 6.9b shows the workflow of each core of the considered multi-core system with Miss Table and victim caches. Each core uses its victim cache to temporarily store the victim blocks from its CL1 and additional blocks from main memory when stream buffering is used. The types of the tasks (like integer or float) and cache hit ratio are used to obtain the mean delay per task and total power consumption.

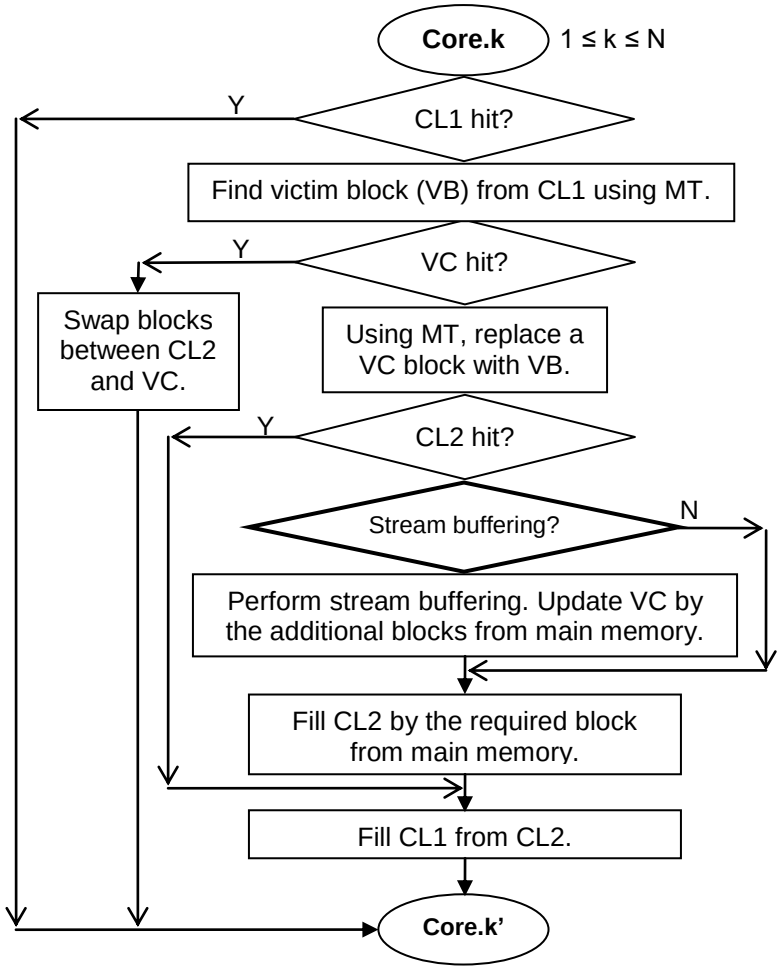


Figure 6.9b: Workflow diagram of each core in the proposed scheme using Miss Table and victim caches in an N-core system.

Table 6.3 shows the relevant system parameters used in this study. These parameters are typical for studying currently available processors such as Intel Xeon.

Table 6.3: System parameters and their values

Parameters	Values
Hit level-1 cache	1 processor clock
Hit victim cache	2 processor clock
Hit level-2 cache	3 processor clock
Stream buffering / bus delay	10 processor clock
Fill level-1 cache buffer	1 processor clock
Fill victim cache buffer	1 processor clock
Fill level-2 cache buffer	1 processor clock

6.7.4 Simulation Tools and Workload

In this chapter, we present a Miss Table based cache locking scheme with victim caches to improve the predictability and performance/power ratio. We develop a simulation platform using two popular simulation tools – Heptane and VisualSim. The simulation platform includes Heptane in Fedora 10 operating system and VisualSim in Windows XP operating system in a Dell PowerEdge 1600SC PC.

Heptane (Hades Embedded Processor Timing ANalyzEr) is a prominent worst case execution time (WCET) analyzer from IRISA, a research unit in the forefront of information and communication science and technology [162]. Heptane simulates a processing core, takes C code as the input application, and generates tree-graph that shows the blocks that cause misses. After post-processing the tree-graph, block addresses are selected for the Miss Table. We use Heptane to characterize the workload that should be used to run VisualSim simulation program. Input and output parameters for Heptane are shown in Table 6.4.

Table 6.4: Input/output parameters for Heptane

Input	Output
Application: C code XML file (for locking)	Block address Number of misses Number of instructions

VisualSim (short for VisualSim Architect) is an outstanding system-level simulation tool from Mirabilis Design [172]. VisualSim provides a graphic interface to model and simulate real-time embedded multi-core systems running multimedia applications. We use VisualSim to simulate our proposed Miss Table based cache locking scheme with victim caches in a multi-core system.

Input and output parameters for VisualSim are shown in Table 6.5.

Table 6.5: Input/output parameters for VisualSim

Inputs	Outputs
Number of integers Number of floats Number of loads/stores Number of branches Number of tasks Cache miss rate Miss Table	Mean delay per task Total power consumption

In this work, we use Moving Picture Experts Group's MPEG4 (part-2), Advanced Video Coding – widely known as H.264/AVC, Fast Fourier Transform (FFT), Matrix Inversion (MI), and Discrete Fourier Transform (DFT) applications to run the simulation program. Important information about these files is shown in Table 6.6. The number of instructions is obtained from Heptane analysis.

Table 6.6: Some statistics of MPEG4, H.264/AVC, FFT, MI, and DFT applications

Applications	Code Size (KB)	Number of Instructions
MPEG4	29.94	5,207,118
H.264/AVC	22.45	3,905,338
FFT	2.34	365,184
MI	1.47	227,518
DFT	1.16	171,307

Important input parameters used in the simulation are shown in Table 6.7. We obtain results for various I1 (/D1) cache size, line size, and associativity level with and without applying cache locking and victim caches.

Table 6.7: Simulation input parameters and their values

Parameters	Values
I1 (/D1) cache size (KB)	2, 4, 8, 16, or 32
CL1/CL2 line size (Byte)	16, 32, 64, 128, or 256
CL1/CL2 associativity level	1-, 2-, 4-, 8-, or 16-way
CL2 cache size (KB)	256 (fixed)
Cache Locking	level-2 cache locking (0% to 50%)
Number of cores	4 (fixed)

Output parameters in this work are the mean delay per task and total power consumption. We define delay as the time between the start of execution of a task and its end. We use an activity based power analysis to compare the total power consumed by the system. A system component (core, cache, bus, or main memory) is considered to be in one of the three states – Active (Full-On), Standby (On), or Sleep (Off).

In Active (Full-On) state, a component receives full (adequate) power from the system to deliver full functionality to the user. For a task i, a component j consumes Pij

(active) amount of power to be fully functional. At this state, it runs maximum number of functions and consumes maximum amount power.

In Standby (On) state, a component is partially powered with automatic wakeup on request. For the same task i, the same component j consumes Pij (standby) amount of power while remaining in Standby state.

In Sleep (Off) state, a component is turned off and should not consume any significant energy.

Therefore, the total power consumption by the system is expressed as shown in Equation (1). In this equation, X is the total number of tasks and Y is the total number of components.

$$Pt\ (total) = \sum_{i=1,\ j=1}^{X,\ Y} \Sigma\ (Pij\ (active) + Pij\ (standby) \qquad \text{Equation (1)}$$

In this work, we use the power model provided by VisualSim [172] to obtain the total power consumption. For multi-core systems with many cores, power model with more states is complicated and detrimental (something that can be studied further).

6.7.5 Experimental Results

In this experiment, we evaluate the impact of Miss Table based cache locking with victim caches on the predictability and performance/power ratio for real-time embedded systems. We model a system with 4 cores and run the simulation program using MPEG4, H.264/AVC, FFT, MI, and DFT workload. We obtain results by varying the amount of level-2 locked cache size, I1 cache size, I1 line size, and I1 associativity level. We present the simulation results in the following subsections.

Previously (in Chapter 5), we have seen that 25% cache locking produces the optimal performance and predictability. In this chapter, we apply 25% cache locking to show the impact of cache locking. The mean delay per task versus I1 cache size for no locking and 25% locking is shown in Figure 6.10. Experimental results show that mean delay per task for MPEG4 and H.264/AVC decreases when we move from no locking to 25% locking and/or from smaller I1 to larger I1; the decrement is significant for smaller I1. However, mean delay per task for FFT, MI, and DFT remains the same when we move from no locking to 25% locking and from smaller I1 to larger I1. This is because FFT, MI, and DFT code entirely fit into I1. But MPEG4 and H.264/AVC applications are bigger than those of FFT, MI, or DFT and do not entirely fit into I1. Results also show that mean delay per task due to MPEG4 is always greater than those of others. This is because MPEG4 has heavier workload than the others' workload.

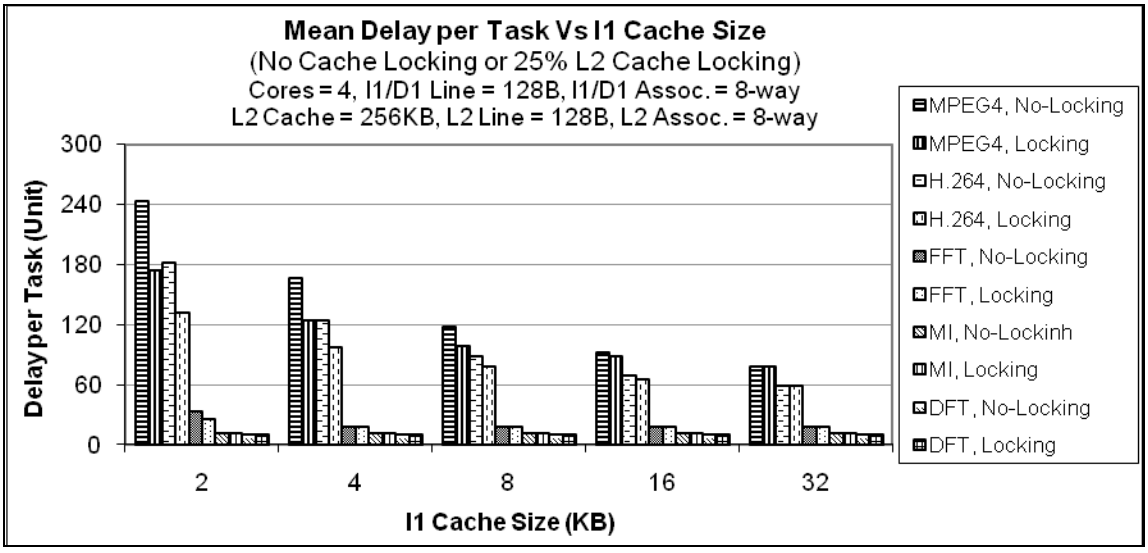


Figure 6.10: Mean delay per task versus I1 cache size.

Similar behavior is observed for the total power consumption versus I1 cache size for no locking and 25% locking (see Figure 6.11). In this experiment, we perform activity based power analysis – the more the cache is active, the more power it (and the system) consumes. Simulation results show that total power consumption decreases when we move from no locking to 25% locking and/or from smaller I1 to larger I1 for MPEG4 and H.264/AVC; the decrement is significant for smaller I1. However, total power consumption for FFT, MI, and DFT remains almost the same when we move from no locking to 25% locking and from smaller I1 to larger I1. Again, this is because the code of FFT, MI, and DFT entirely fit into I1 but the code of MPEG4 and H.264/AVC do not entirely fit into I1. Like mean delay per task, results also show that total power consumption due to MPEG4 is always greater than those of others, because MPEG4 has heavier workload than the others' workload.

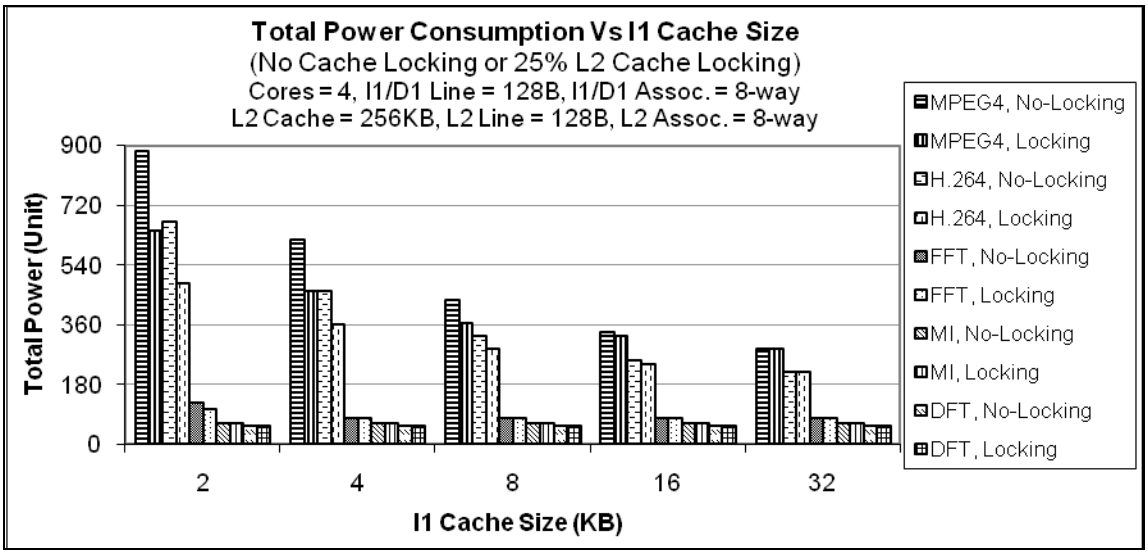


Figure 6.11: Total power consumption versus I1 cache size.

The results due to FFT, MI and DFT applications are very similar; so in the rest of the discussion, the FFT results will also represent the omitted MI and DFT results. Similarly, the MPEG4 results will also represent the omitted H.264/AVC results.

The average delay per task versus I1 line size for no locking and 25% locking is shown in Figure 6.12. We notice that the average delay per task goes down for MPEG4, regardless of the line size when cache locking is used. For MPEG4, mean delay per task decreases with the increase of line size when line size is smaller than 128B, but mean delay per task increases with the increase of line size when line size is larger than 128B. However, the average delay per task for FFT remains the same. This is because FFT code fits entirely in 4 KB I1, so changing line size and/or applying cache locking do not impact mean delay per task.

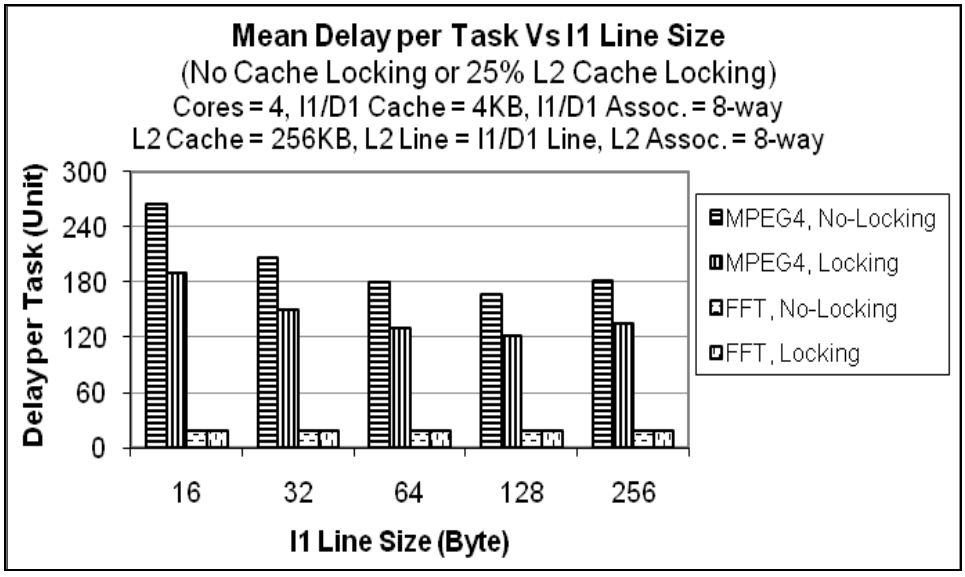


Figure 6.12: Mean delay per task versus I1 line size.

Similarly, we notice that 25% cache locking helps decrease total power consumption regardless of I1 line size [see Figure 6.13]. It is also noted that total power consumption decreases with increasing I1 line size leveling off at a line size of 128B. Again, total power consumption for FFT remains the same because FFT code fits entirely in 4 KB I1 and changing line size and/or applying cache locking do not impact on mean delay per task.

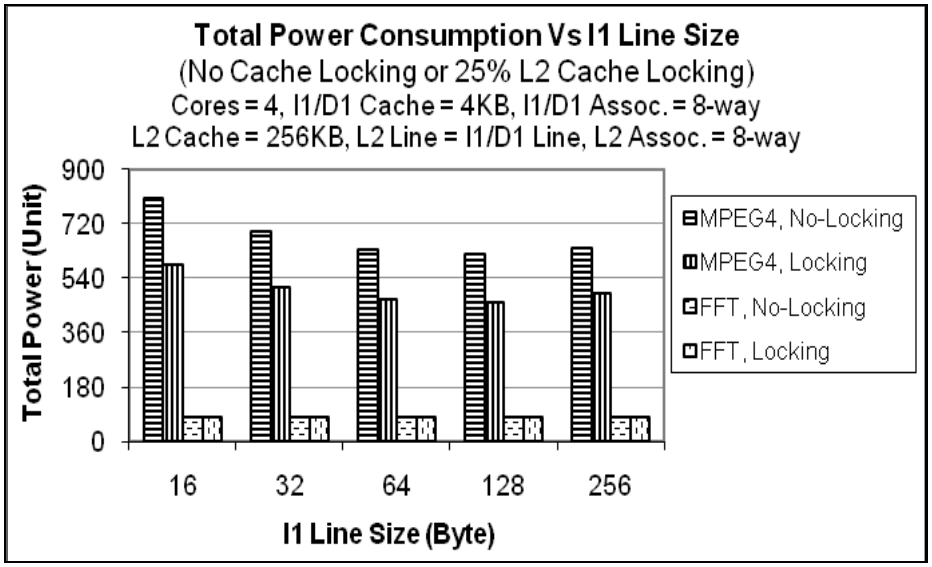


Figure 6.13: Total power consumption versus I1 line size.

The impact of no cache locking and 25% locking on mean delay by varying I1 associativity level is shown in Figure 6.14. Experimental results show that for any I1 associativity level, mean delay per task for MPEG4 decreases when we move from no locking to 25% locking. Also, the mean delay per task decreases with the increase in I1 associativity level. The decrease is significant for smaller associativity levels. For 4 KB I1, mean delay per task for FFT remains the same when associativity level is changed and/or cache is used.

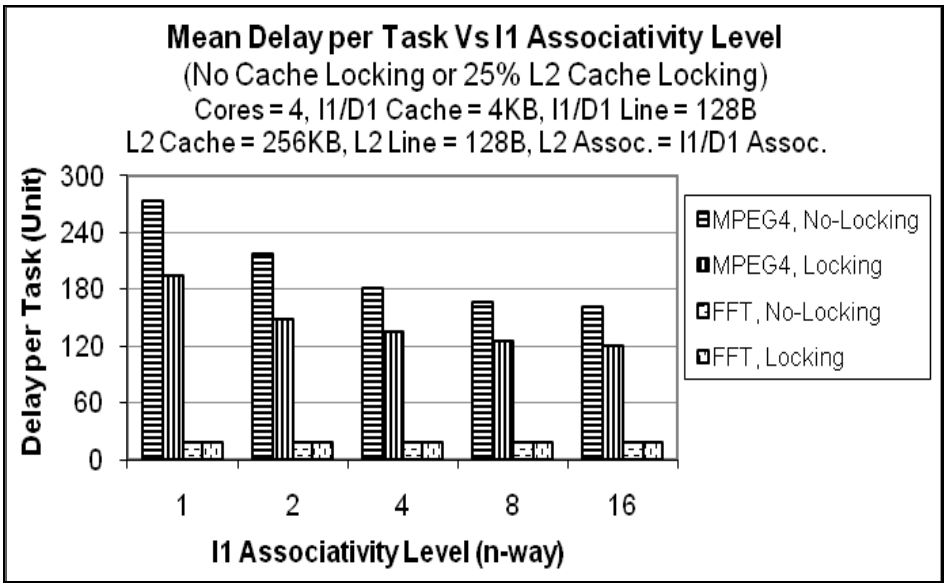


Figure 6.14: Mean delay per task versus I1 associativity level.

Figure 6.15 illustrates the impact of no cache locking and 25% locking on total power consumption by varying I1 associativity level. Experimental results show that for any I1 associativity level, total power consumption for MPEG4 decreases when we move from no locking to 25% cache locking. Also, total power consumption decreases with the increase in I1 associativity level. Again, total power consumption for FFT remains the same for 4 KB I1.

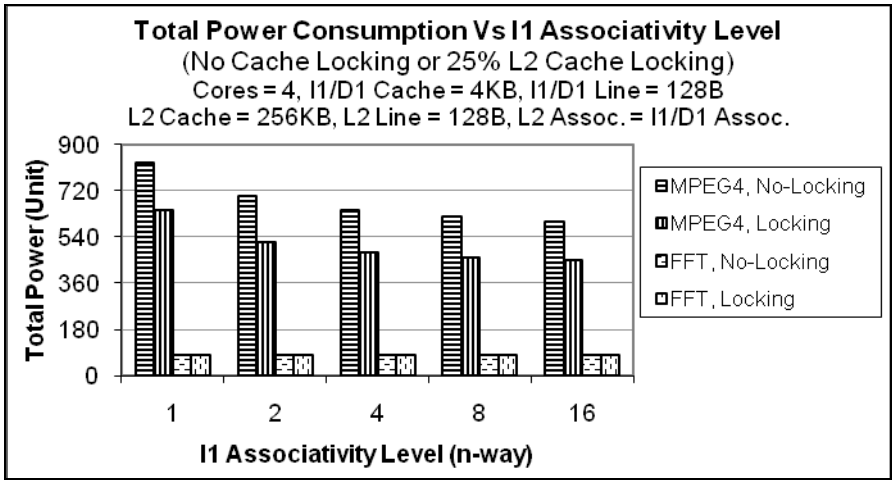


Figure 6.15: Total power consumption versus I1 associativity level.

Finally, we present the impact of using the Miss Table and victim caches with cache locking on mean delay per task and total power consumption. As we know that with the increase in the number of locked blocks – on the one hand, cache blocks that might cause most of the misses are locked; but on the other hand, the effective cache size decreases. Miss Table helps improve hit ratio by selecting the most important memory blocks for cache locking and victim caches help improve hit ratio by holding CL1 victim blocks (and additional blocks when stream buffering is used).

Experimental results show that using Miss Table with cache locking decreases mean delay per task for MPEG4 application. From Figure 6.16, it is observed that mean delay per task starts decreasing with the increase in the number of locked blocks for MPEG4 application. It is also observed that mean delay per task decreases for all amount of cache locking when victim caches are used. Beyond 25% cache locking (for MPEG4), the mean delay per task increases with the increase in the number of locked blocks. Simulation results also show that when smaller applications like FFT entirely fits in I1, there is no positive impact of cache locking on mean delay per task.

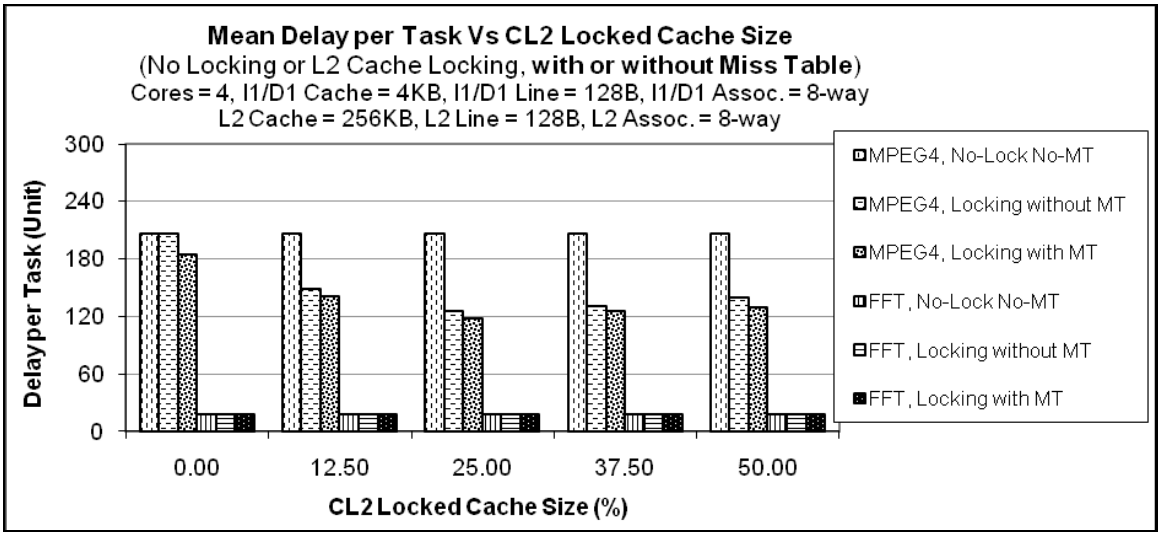


Figure 6.16: Mean delay per task versus level-2 locked cache size (with and without Miss Table MT).

Using Miss Table with cache locking has positive impact on total power consumptions for some applications. From experimental results we notice that total power consumption starts decreasing with the increase in the number of the locked blocks for MPEG4 application (see Figure 6.17). We also notice that total power consumption decreases for all amount of cache locking when Miss Table is used with cache locking (up to 25% cache locking for MPEG4). However, for smaller applications like FFT, there is no positive impact of cache locking on total power consumption as they entirely fit in 4KB I1 cache.

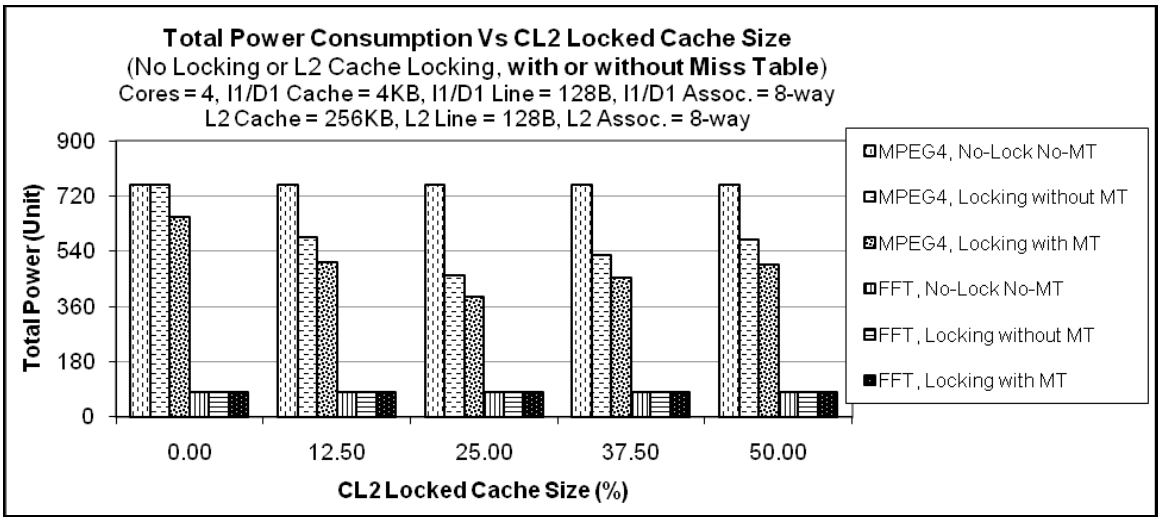


Figure 6.17: Total power consumption versus level-2 locked cache size (with and without Miss Table MT).

Finally, we present the impact of using victim caches in MT based cache locking scheme on mean delay per task and total power consumption. Experimental results reveal that for MPEG4 application, mean delay per task decreases for all amount of cache locking when victim caches are used (see Figure 6.18). Beyond 25% cache locking (for MPEG4), the mean delay per task increases with the increase in the number of locked blocks. Simulation results also show that when FFT application entirely fits in 4KB I1, there is no positive impact of using victim caches on mean delay per task.

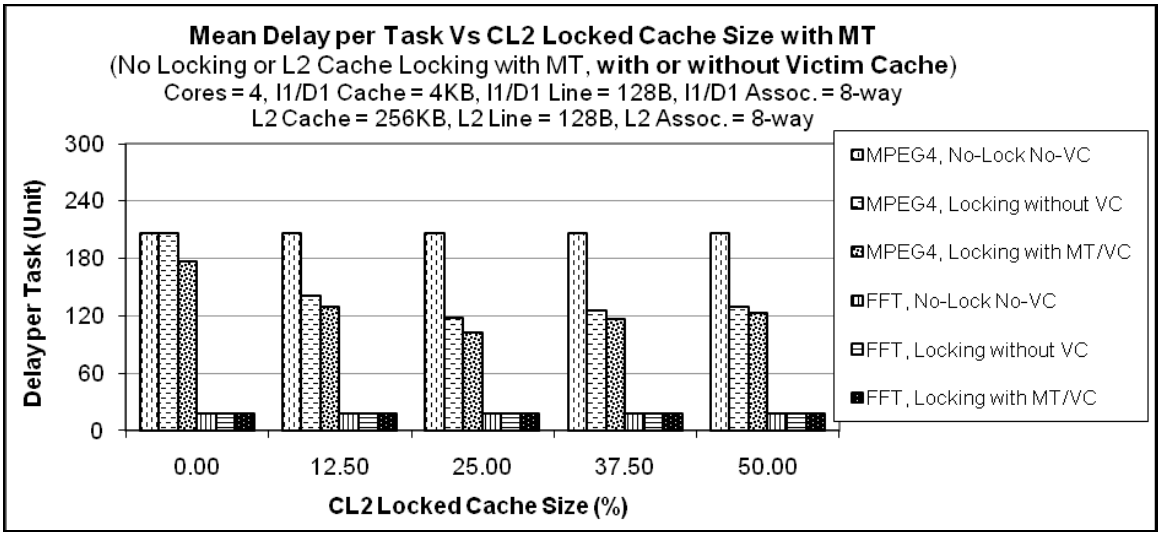


Figure 6.18: Mean delay per task versus level-2 locked cache size (with MT; with and without victim cache VC).

Experimental results also reveal that total power consumption starts decreasing with the increase in the number of the locked blocks for MPEG4 application as shown in Figure 6.19. We also notice that total power consumption decreases for all amount of cache locking when victim caches are used. Beyond 25% cache locking (for MPEG4),

total power consumption increases with the increase in the number of locked blocks. Again, when applications entirely fits in I1 (as FFT fits in 4KB I1), there is no positive impact of using victim caches on total power consumption.

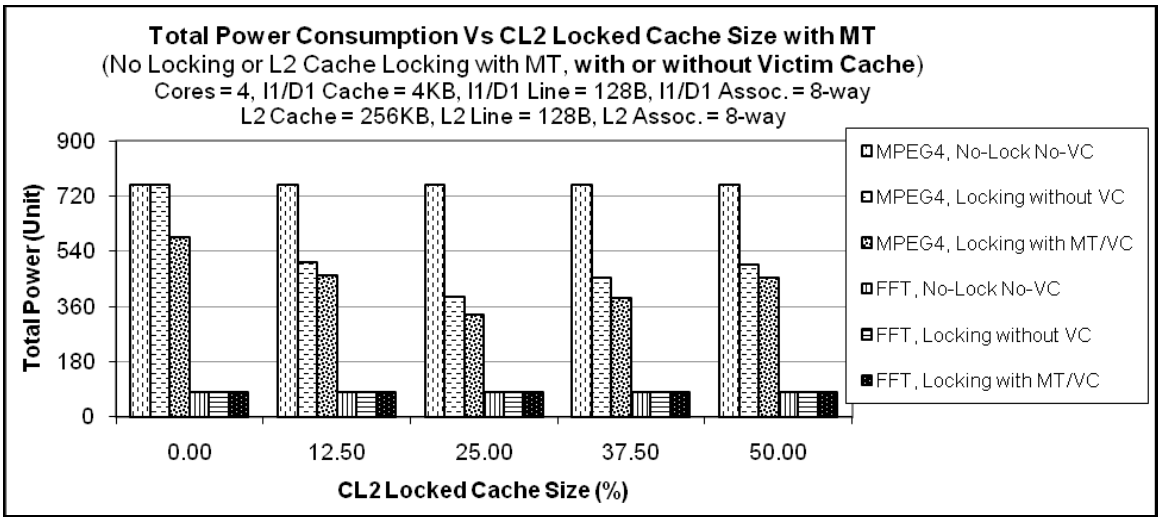


Figure 6.19: Total power consumption versus level-2 locked cache size (with MT; with and without victim cache VC).

6.8 Summary

Execution time predictability is an essential factor for designing real-time systems. Recently designed multi-core processors run at lower frequency and produce high performance/power ratio. However, multilevel caches in multi-core processors make the execution time predictability very difficult. It has been proven that cache locking improves predictability for single-core systems. In this chapter, we improve predictability of a real-time embedded system developing Miss Table based cache locking with victim caches. We simulate an 4-core system. It should be noted that this cache locking

technique is applicable for single-core cache locking, too. The simulated architecture has two-level cache memory hierarchy. We generate MPEG4, H.264/AVC, FFT, MI, and DFT workloads by post-processing their respective tree-graphs created by WCET analysis. A simulation platform is developed to simulate cache locking in multi-core systems. The impact of Miss Table and victim caches on mean delay per task and total power consumption is summarized in Table 6.8. The percent changes, decrement (-) or increment (+), in mean delay per task and total power consumption are relative to the values obtained without using Miss Table and victim caches (see Figures 6.16 – 6.19).

Table 6.8: Changes (in %) of delay and power for MPEG4 and FFT applications

	Application	**Percentage Changes in Delay and Power**			
		Non-Locking		**25% CL2 locking**	
		Delay	**Power**	**Delay**	**Power**
Miss Table	MPEG4	(-)11	(-)17	(-)25	(-)32
	FFT	0	0	0	0
Miss Table and Victim Caches	MPEG4	(-)15	(-)22	(-)33	(-)41
	FFT	0	0	0	0

For MPEG4, 33% reduction in mean delay per task and 41% reduction in total power consumption are achieved using Miss Table and victim caches. However, using Miss Table and victim caches has no positive impact on performance/power ratio for FFT when I1 size 4KB or larger. Experimental results show that this method may improve execution time predictability by reducing cache misses by more than 50% for 25% cache locking (see Table 6.2). Additional predictability enhancement is possible by trading off mean delay per task and total power consumption.

CHAPTER 7

CONCLUSION AND FUTURE WORK

Cache improves performance by reducing the speed gap between the main memory and CPU. However, cache poses challenges to real-time embedded systems as it introduces execution time unpredictability and consumes huge amount of power to be operated. In this dissertation, we develop a methodology to optimize cache for real-time embedded systems. We improve performance and decrease power consumption by optimizing cache parameters. We enhance predictability by applying cache locking. We introduce Miss Table at cache level to improve cache locking performance and use victim cache(s) to increase cache hits. Simulation results show that Miss Table based cache locking with victim cache(s) improves predictability and performance/power ratio. This chapter concludes this dissertation and discusses some important extensions of this work.

7.1 Conclusion

Cache is an essential component in every modern computing system to improve the performance by bridging the main memory and CPU speeds. However, cache increases execution time unpredictability due to its adaptive and dynamic nature. Also, cache consumes vast amount of energy due to the fact that it is power-hungry. Execution time predictability is a crucial factor for the success of real-time systems. Similarly,

energy requirement is crucial for embedded systems as they suffer from limited resources. Studies show that cache parameters have significant influences on the performance and power consumption of embedded applications. However, techniques used to improve performance/power ratio may worsen the predictability. Similarly, techniques used to improve the predictability may worsen performance/power ratio. For modern real-time embedded systems, the performance, power consumption, and predictability – all are important. Existing solutions do not address the performance, power consumption, and predictability issues together and are not very effective for multi-core architecture. In this work, we develop a cache optimization methodology to analyze and improve the predictability and performance/power ratio of real-time embedded systems at the same time.

First, we develop a cache modeling and optimization technique to enhance the performance of a single-core system. Cache size, line size, associativity level, and cache levels are optimized to improve the performance. Simulation results indicate that the performance of embedded systems can be enhanced by cache modeling and optimization for real-time applications. This cache modeling and optimization technique can be used to analyze embedded system architectures and determine the optimal cache parameters for target applications.

Second, we develop a methodology to improve performance/power ratio of multi-core embedded systems using cache optimization. Experimental results, using an 2-core architecture running MPEG4 multimedia application, show that utilization, mean delay, number of transactions, and total power consumption can be optimized by changing the cache size. This cache optimization technique can be used to explore multi-core

architecture running real-time multimedia applications and find the optimal cache parameters for the best performance/power ratio.

Third, we introduce a cache locking scheme to improve the predictability of real-time embedded systems. It is important to note that cache locking performance depends on the efficiency and accuracy of the block selection criteria. We introduce an algorithm that selects blocks (to be locked), which might cause most misses in the future (of the same execution) if not locked. Simulation results show that this cache locking technique can drastically improve the predictability in real-time embedded systems.

Forth, we propose Miss Table at cache level to improve cache locking and cache replacement performance. We use victim cache that temporarily stores the victim blocks from CL1 and supports stream buffering to improve performance/power ratio by reducing cache misses. This scheme is suitable for both single-core and multi-core architectures. Simulation results, using an 4-core architecture running real-time applications, show that Miss Table based cache locking scheme with victim caches significantly improves the predictability and performance/power ratio.

Finally, we develop strategies to characterize applications and generate useful workload to evaluate cache optimization methodologies. To evaluate our proposed Miss Table based cache locking with victim caches in a multi-core system we develop a technique that has 3 major phases. In phase-I, we divide large application code into smaller end-to-end functions. In phase-II, we estimate major operations (integer, floating-point, load/store, and branch) in the code segment. Finally in phase-III, we select the cache blocks that should be locked. We find the workload very useful.

Proposed Miss Table at cache level improves the performance of cache locking and cache replacement policy. Victim cache improves hit ratio by storing victim blocks from level-1 cache and supporting stream buffering. Experimental results, using an 4-core architecture, show that this cache locking method may improve execution time predictability by reducing cache misses by more than 50% when locking 25% of the cache size. In this experiment, proposed Miss Table based cache locking with victim caches has significant impact on performance/power ratio. For MPEG4, 33% reduction in mean delay per task and 41% reduction in total power consumption are achieved using Miss Table and victim caches. However, for small applications like FFT, when the code entirely fits in the cache, there is no positive impact of cache locking, Miss Table, and victim caches on predictability and performance/power ratio.

7.2 Future Extensions

Cache optimization is a fundamental area of interest to the computer engineers, scientists, and researchers. As future computing architectures and systems change, cache memory organization needs to be analyzed and optimized to achieve the optimal performance, power consumption, and predictability for the target applications. From current design trend, multi-core architecture is the future of all computing systems. Due to increasing demands and resource constrains, real-time embedded systems need to be designed with special care to meet the future requirements. Our dissertation contributions including the simulation platforms can be extended to cope with the following important research areas.

Evaluate cache locking at various levels of caches: In this dissertation, we develop simulation platforms to implement cache locking at level-1 instruction cache in a single-core architecture (see Chapter 5) and cache locking at level-2 shared cache in a multi-core architecture (see Chapter 6). This work can be extended to evaluate cache locking at level-1 data cache (in single-core and multi-core architectures) and at level-2 private caches in multi-core architectures.

Investigate data cache locking for embedded systems: Like level-1 instruction cache, level-1 data cache also has significant impact on the performance, power consumption, and execution time predictability. It is found that for multimedia applications there is sufficient reuse of values for caching [121]. Unlike single-core architectures, data consistency and concurrency become more challenging in multi-core architectures. This work can be extended to investigate cache locking at level-1 data cache in both single-core and multi-core embedded systems.

Explore power-aware multi-core architecture: Due to its tremendous potential, multi-core architecture is being deployed in all sorts of modern computing devices. Power consumption and dissipation are extremely crucial for complex architectures with thousands of cores. The maximum number of cores is expected to be active to achieve the best performance. However, the more cores are active, the more power will be consumed (and dissipated). Traditionally, the shortest path between the source and the destination is the best path; which may not be beneficial for power-aware multi-core systems. Therefore, new core allocation strategies and routing algorithms are required to deal with multi-core architecture issues including performance and power consumption. Our multi-

core cache modeling and simulation platform can be extended to explore power-aware multi-core architecture.

We hope the above discussion motivates the interested scholars into considering research in the challenging and constrained environment of real-time embedded systems. Multi-core architecture is the future of all modern computing areas from desktop to embedded environments. Multi-core architecture comes with multi-level caches. At present, there is a great deal of research in the areas of performance, power consumption, and predictability of multi-core architecture, which makes the prospects of this research work very exciting for real-time embedded systems. Our contributions lead to solutions that overcome the disadvantages due to the presence of caches in multi-core, as well as single-core, embedded systems. Certainly, the potential of caches in real-time embedded systems then will be enormous.

APPENDIXES

Appendix A: Complete FFT source code.

Here is the complete Fast Fourier Transform (FFT) source code, annotated the way Heptane needs it. Total number of Integer, Floating-point, Load/Store, and Branch operations shown in Table 6.1 are calculated for this code. Also, block address and total cache misses for that block shown in Table 6.2 are obtained from the Haptane tree-graph generated using this code.

```
Unsigned NumberOfBitsNeeded (unsigned PowerOfTwo)
{
   Unsigned i, res;
   for (i = 0; PowerOfTwo == 1; i++) [11] {
      PowerOfTwo = PowerOfTwo / 2;
      res = i;
   }
   return res;
}
unsigned ReverseBits (unsigned index, unsigned NumBits)
{
   unsigned       i, rev;
   for (i = rev = 0; i < NumBits; i++) [10] {
      rev = (rev << 1) | (index & 1);
      index = (index >> 1);
   }
   return rev;
}

fft()
{
   unsigned       NumSamples = 2048;
   double         RealIn [2048];
   double         ImagIn [2048];
   double         RealOut[2048];
   double         ImagOut[2048];
```

```
    double          SINTAB [11] [2];
    unsigned        NumBits;
    unsigned        i, j, sin_index, k, n;
    unsigned        BlockSize, BlockEnd;
    double          angle_numerator = 2.0 * (3.14159265358979323846);
    double          delta_angle;
    double          alpha, beta;
    double          delta_ar;
    double          tr, ti;
    double          ar, ai;
    SINTAB [0] [0] = 1.0;
    SINTAB [10] [1] = 0.003068;

    NumBits = NumberOfBitsNeeded (NumSamples);

    for (i = 0; i < NumSamples; i++) [2048] {
       j = ReverseBits(i, NumBits);
       RealOut[j] = RealIn[i];
       ImagOut[j] = ImagIn[i];
    }

    BlockEnd = 1;
    sin_index = 0;

    for(BlockSize = 2; BlockSize <= NumSamples; BlockSize = BlockSize << 1)
    [11, pow (2, (i + 1))] {
       delta_angle = angle_numerator / (double) BlockSize;
       alpha = SINTAB [sin_index] [0];
       alpha = 2.0 * alpha * alpha;
       beta = SINTAB [sin_index] [1];
       sin_index++;

       for (i = 0; i < NumSamples; i += BlockSize) [(2048 / nlast (P, 1))] {
          ar = 1.0;
          ai = 0.0;

          for (j = i, n = 0; n < BlockEnd; j++, n++) [nlast (P, 2) / 2] {
             k = j + BlockEnd;
             tr = ar * RealOut[k] - ai * ImagOut[k];
             ti = ar * ImagOut[k] + ai * RealOut[k];
             RealOut[k] = RealOut[j] - tr;
             ImagOut[k] = ImagOut[j] - ti;
             RealOut[j] += tr;
             ImagOut[j] += ti;
             delta_ar = alpha * ar + beta * ai;
             ai -= (alpha * ai - beta * ar);
             ar -= delta_ar;
          }
       }
       BlockEnd = BlockSize;
    }
}
```

Appendix B: Heptane generated Tree-graph for FFT source code.

Heptane generated syntax tree-graph for First Fourier Transform (FFT) source code is shown in Figure A. A syntax tree is a tree whose nodes represent the structure of programs in the high-level language like C and whose leaves represent basic blocks. Leaves (example: CALL, CODE) in the syntax tree may coincide with the control-flow nodes (example: SEQ, LOOP).

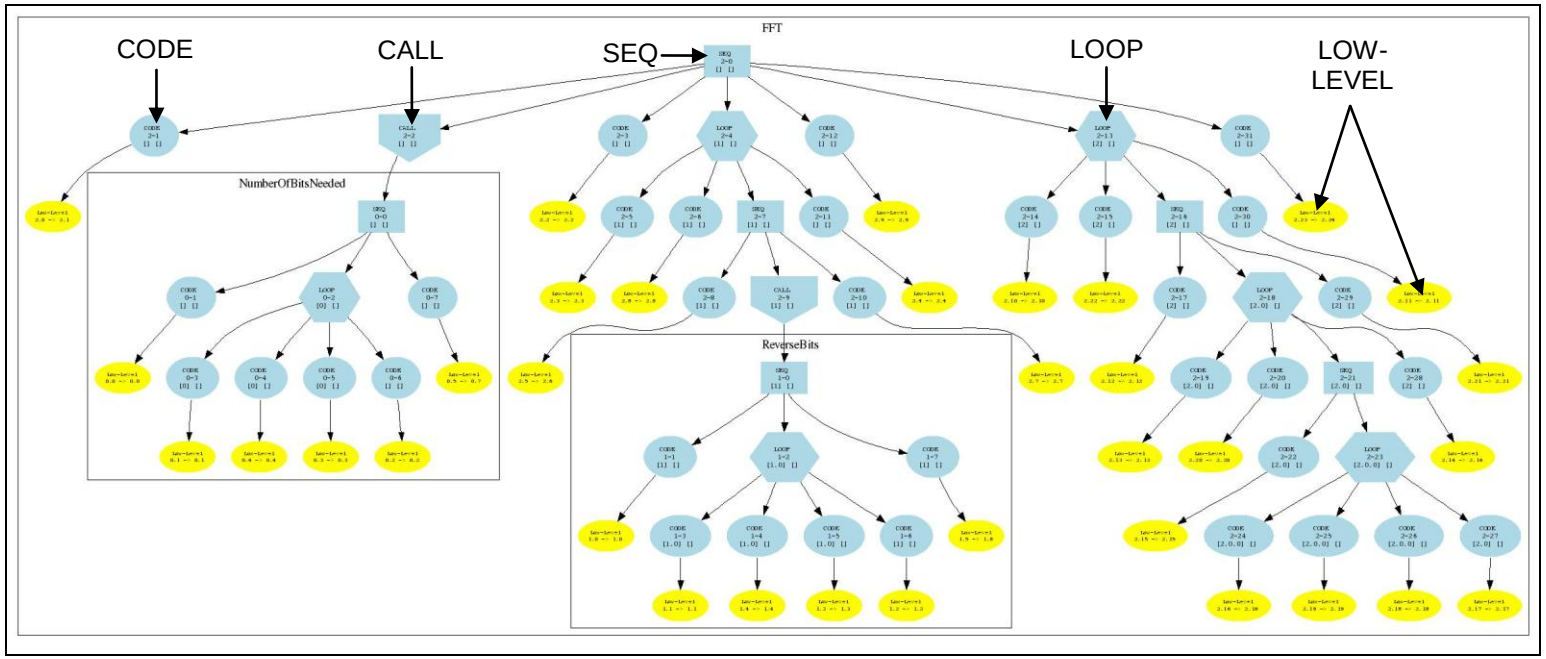


Figure A. Syntax tree-graph for FFT application.

Contextual information regarding leaves and control-flow nodes (CODE, LOOP, etc) can be obtained by clicking on the nodes. Contextual information include the following,

a) Number of instructions – total number of instructions regarding that node
b) Number of cycles – total number of cycles required to execute the instructions
c) Cache hit – total number of cache hits
d) Cache miss – total number of cache misses

The information of a LOW-LEVEL node is decomposed into 3 main areas:

a) Memory – contains tables for all cache prediction mechanisms.
b) Branch Target Buffer (BTB) – contains tables for all branch prediction mechanisms and the corresponding analysis, if this was done.
c) Execution – contains tables for all pipelined or un-pipelined executions, plus overlapping pipelines.

In this work, we consider the number of instructions and cache misses for all CODE nodes. As shown in the tree-graph, a LOOP node represents a set of CODE nodes. So, the number of instructions and cache misses for loop nodes are excluded. Also, the number of instructions and cache misses for SEQ nodes are excluded as they are negligible.

BIBLIOGRAPHY

[1] J. Absar and F. Catthoor. Analysis of Scratch-Pad and Data-Cache Performance Using Statistical Methods; IEEE, pages 820-825, 2006.

[2] R.E. Ahmed. Energy-Aware Cache Coherence Protocol for Chip-Multiprocessors. Canadian Conference on Electrical and Computer Engineering (CCECE-2006), pages 82-85, 2006.

[3] D. Ait-Boudaoud, M.K. Ibrahim, and B.R. Hayes-Gill. Novel cell architecture for bit level systolic arrays multiplication. IEEE Proceedings - Computers and Digital Techniques, 1991.

[4] G. Albera and R.I. Bahar. Power/Performance Advantages of Victim Buffer in High-Performance Processors. IEEE Volta International Workshop on Low Power Design. Como, Italy, 1999.

[5] A. Asaduzzaman and I. Mahgoub. Cache Modeling and Optimization for Portable Devices Running MPEG-4 Video Decoder. The international journal of Multimedia Tools and Applications (MTAP), A special issue of MTAP, 2006.

[6] A. Asaduzzaman and I. Mahgoub. Cache Optimization for Embedded Systems Running H.264/AVC Video Decoder; AICCSA06 IEEE International Conference, UAE, pages 665-672, 2006.

[7] A. Asaduzzaman and I. Mahgoub. Evaluation of Application-Specific Multiprocessor Mobile System. SPECTS'04, ISBN: 1-56555-284-9, pages 751-758, San Jose, CA, 2004.

[8] A. Asaduzzaman and I. Mahgoub. Impact of Tuning Cache Parameters on Embedded MPEG4 and H.264/AVC CODEC, Submitted for EURASIP, 2009.

[9] A. Asaduzzaman, I. Mahgoub, and F.N. Sibai. Impact of L1 Entire Locking and L2 Way Locking on Performance, Power Consumption, and Predictability of Multicore Real-Time Systems. 7th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-2009), Rabat, Morocco, 2009.

[10] A. Asaduzzaman, I. Mahgoub, et al. Cache Optimization for Mobile Devices Running Multimedia Applications, ISMSE-2004, FL, 2004.

[11] A. Asaduzzaman, N. Limbachiya, I. Mahgoub, and F. Sibai. Evaluation of ICache Locking Technique for Real-Time Embedded Systems. IEEE/IIT-2007, UAE, 2007.

[12] A. Arnaud and I. Puaut. Dynamic Instruction Cache Locking in Hard Real-Time Systems. http://www.irisa.fr/caps/publications/pdfs/arnaud-Locking.pdf, 2005.

[13] A. Avritzer, J. Kondek, D. Liu, and E.J. Weyuker. Software Performance Testing Based on Workload Characterization; WOSP-2002, Rome, Italy, pages 17–24, 2002.

[14] M. Ball, C. Cifuentes, and D. Bairagi. Partitioning of Code for a Massively Parallel Machine. Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. pages 225-236, 2004.

[15] L. Benini and G.D. Micheli. Networks on Chip: A New Paradigm for Systems on Chip Design. IEEE, 2002.

[16] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. Circuits and Systems Magazine, IEEE, 2004.

[17] T. Bjerregaard and S. Mahadevan. A survey of research and practices of Network-on-chip. Source ACM Computing Surveys (CSUR), Volume 38, Issue 1, 2006.

[18] N. Blachford. Cell Architecture Explained – Part 1: Inside The Cell. http://www.blachford.info/computer/Cells/Cell1.html, 2005.

[19] N. Blachford. Cell Architecture Explained Version 2. http://www.blachford.info/computer/Cell/Cell0_v2.html, 2006.

[20] J.A. Brown, R. Kumar, and D. Tullsen. Proximity-Aware Directory-based Coherence for Multi-core Processor Architectures. SPAA-2007, CA, 2007.

[21] A.M. Campoy, A. Pedes, F. Rodriguez, and J.V. Busquets-Mataix. Static Use of Locking Caches in Multitask Preemptive Real-Time Systems. IEEE Real-Time Embedded System Workshop, 2001.

[22] A.M. Campoy, A. Perles, F. Rodriguez, and J.V. Busquets-Mataix. Static Use of Locking Caches Vs. Dynamic use of Locking Caches for Real-Time Systems. IEEE/CCECE-2003, Montreal, 2003.

[23] A.M. Campoy, A.P. Ivars, and J.V. Busquets Mataix. Dynamic Use of Locking Caches in Multitask, Preemptive Real-Time Systems. 15th Triennial World Congress, Barcelona, Spain, 2002.

[24] A.M. Campoy, A.P. Ivars, and J.V. Busquets-Mataix. Using Locking Caches in Preemptive Real-Time Systems. Proceedings of the 12th IEEE Real-Time Congress on Nuclear and Plasma Sciences, pages 157-159, 2001.

[25] A.M. Campoy, A.P. Jimenez, A.P. Ivars, and J.V. Busquets Mataix. Using Genetic Algorithms in Content Selection for Locking-Caches. Proceedings of IASTED International Symposia Applied Informatics, pages 271-276, Austria, 2001.

[26] A.M. Campoy, E. Tamura, S. Saez, F. Rodriguez, and J.V. Busquets-Mataix. On Using Locking Caches in Embedded Real-Time Systems. ICESS-05, LNCS 3820, pages 150-159, 2005.

[27] A.M. Campoy, I. Puaut, A.P. Ivars, and J.V. Busquets-Mataix. Cache Contents Selection for Statically-Locked Instruction Caches: An Algorithm Comparison. Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS-2005), Vol. 00, pages 49-56, 2005.

[28] A.M. Campoy, S. Saez, A. Perles, and J.V. Busquets-Mataix. Performance Analysis of static use of Locking Caches. Proceedings of the 3rd WSEAS Int. Conference on Automation and Information. Tenefire, Spain, 2002.

[29] A.M. Campoy, S. Saez, A. Perles, and J.V. Busquets-Mataix. Performance Comparison of Locking Caches Under Static and Dynamic Schedulers. 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming, Poland, 2003.

[30] B. Cernuschi-Frías, J. Luis-Hamkalo, J.D. Pfefferman, and H. Gonzalez. Analysis of Cache Memory Strategies for Some Image processing Applications. IEEE, 2001.

[31] M. Chidester and A. George. Parallel Simulation of Chip-Multiprocessor Architectures. ACM, pages 176-200, 2002.

[32] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches. DAC/ACM, 2000.

[33] C.J. Choi, G.H. Park, J.H. Lee, W.C. Park, and T.D. Han. Performance Comparison of Various Cache Systems for Texture Mapping. The Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, pages 374-379, 2000.

[34] M. Chynoweth, M.R. Lee. Implementing Scalable Atomic Locks for Multi-Core Intel® EM64T and IA32 Architectures. http://www.intel.com/cd/ids/developer/asmona/eng/dc/threading/333935.htm, 2005.

[35] R.P. Cook, C.J. Linn, J.L. Linn, and T.M. Walker. Cache memories: A Tutorial and Survey of Current Research Directions. ACM-1982, pages 99–110, 1982.

[36] R. Cucchiara, A. Prati, and M. Piccardi. Data-Type Dependent Cache Pre-Fetching for MPEG Applications. 21st IEEE International Conference on Performance, Computing, and Communications, pages 115-122, 2002.

[37] D. Decotigny and I. Puaut. ARTISST: An Extensible and Modular Simulation Tool for Real-Time Systems. IRISA, France, 2002.

[38] T. DelSole. Predictability and Information Theory - Part I: Measures of Predictability. Center for Ocean-Land-Atmosphere Studies, George Mason University, 2003.

[39] K. Diefendorff and P.K. Dubey. How Multimedia Workloads Will Change Processor Design. IEEE Computer Society Press, Vol. 30, Issue 9, pages 43-45, 1997.

[40] A. Djabelkhir and A. Seznec. Characterization of embedded applications for decoupled processor architecture. IRISA, Compus de Beaulieu, France, 2005.

[41] H. Dybdahl. Architectural Techniques to Improve Cache Utilization. PhD Thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, Norway, 2007.

[42] S. Ely. MPEG video coding - A simple introduction, in EBU Technical Review Winter, 1995.

[43] D.K. Every. IBM's Cell Processor: The next generation of computing?. Shareware Press. http://www.mymac.com/fileupload/CellProcessor.pdf, 2005.

[44] S. Fang, K. Lin, J. Tsai, and Q. Yu. Final Project. Super-Scalar. Stream Buffer. Victim Cache. UCB. www.cs.berkeley.edu/~kubitron/courses/cs152-S01/projects/submit/project7_report2.doc, 2006.

[45] H. Fareed and M. Bassiouni. Adaptive Object Cache Pre-fetching Scheme based on Object flow. IEEE-2005, pages 311-319, 2005.

[46] G. Frick, K.D. Muller-Glaser, and M. Kuhl. A Design Methodology for Distributed Embedded Systems in Industrial Automation. FZI Forschungszentrum Informatik, Germany, 2004.

[47] P. Gepner, and M.F. Kowalik. Multi-Core Processors: New Way to Achieve High System Performance. Proceedings of the International Symposium on Parallel Computing in Electrical Engineering (PARELEC-2006), pages 9-13, 2006.

[48] T. Givargis, F. Vahid, and J. Henkel. Fast Cache and Bus Power Estimation for Parameterized System-on-a-Chip Design. 2005.

[49] M. Grigoriadou, M. Toula, and E. Kanidis. Design and Evaluation of a Cache Memory Simulation Program. IEEE, 2003.

[50] A. Hajare. Performance Modeling of a Multiprocessor Bus Architecture. IEEE, 1991.

[51] M.D. Hill and A.J. Smith. Experimental Evaluation of On-Chip Microprocessor Cache Memories. IEEE, 1984.

[52] C. Hong, K. Park, and Y.T. Song. Hardware Support: A Cache Lock Mechanism without Retry. SNPD/SAWN-2005, pages 44-49, 2005.

[53] J. Hu and R. Marculescu. Energy-Aware Mapping for Tile-based NoC Architectures under Performance Constraints. http://citeseer.ist.psu.edu/569099.html, 2003.

[54] S.H. Hwang, and G.S. Choi. On-Chip Cache Memory Resilience. EE Dept, TX A&M Univ., 2005.

[55] M.M. Jamali, M.M. Hussain, and G.A. Jullien. A signal processing cell architecture; Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP-1987, 1987.

[56] A. Janapsatya, A. Ignjatovic, and S. Parameswaran. Finding Optimal L1 Cache Configuration for Embedded Systems. Proceedings of the 2006 conference on Asia South Pacific design automation, Japan, pages 796-801, 2006.

[57] A. Jerraya, H. Tenhunen, and W. Wolf. Multiprocessor Systems-on-Chips. IEEE Computer Society, pages 36–40, 2005.

[58] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. Western Research Laboratory (WRL), Digital Equipment Corporation, 1990.

[59] K.M. Kavi. Cache Memories. University of Alabama in Huntsville, AL, 2007.

[60] T. Kogel, M. Doerper, et al. A Modular Simulation Framework for Architectural Exploration of On-Chip Interconnection Networks. ACM CODES+ISSS-2003, Newport Beach, CA, pages 7–12, 2003.

[61] D. Koivisto. What Amdahl's Law can tell us about multicores and multiprocessing. http://www.embedded.com/columns/technicalinsights/173603024?_requestid=49279, 2005.

[62] P.J. Koopman. Embedded System Design Issues (the Rest of the Story). Proceedings of the International Conference on Computer Design (ICCD-1996), 1996.

[63] S. Kumar, A. Jantsch, J.-P. Soininen, and A. Hemani. A network on chip architecture and design methodology. IEEE Computer Society Annual Symposium on VLSI, pages 105-112, 2002.

[64] A.I.C. Lai and C.L. Lei. Data Pre-fetching for Distributed Shared Memory Systems. HICSS-1996, V. 1, pages 102-110, 1996.

[65] D. Lenoski, J. Laudon, M.S. Lam, et al. The Stanford Dash Multiprocessor. IEEE, 1992.

[66] M. Levy. Evaluation Digital Entertainment System Performance. IEEE Computer Society, pages 68–72, 2005.

[67] Y. Li and J. Henkel. A framework for estimating and minimizing energy dissipation of embedded HW/SW systems. Proc. 35th Design Automation Conf., pages 188-194, 1998.

[68] J. Liedtke, H. Hartig, and M. Hohmuth. OS-Controlled Cache Predictability for Real-Time Systems. 3rd IEEE Real-Time Technology and Applications Symposium (RTAS-1997), Montreal, Canada, 1997.

[69] M. Loghi and M. Poncino. Exploring Energy/Performance Tradeoffs in Shared Memory MPSoCs: Snoop-Based Cache Coherence vs. Software Solutions. Proceedings of the conference on Design, Automation and Test in Europe, pages 508-513, 2005.

[70] M. Loghi, M. Poncino, and L. Benini. Cache coherence tradeoffs in sharedmemory MPSOCs. ACM Transaction on Embedded Computing Systems, Vol. 5, No. 2, pages 383-407, 2006.

[71] D. Lyonnard, S. Yoo, A. Baghdadi, and A. Jerraya. Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip. DAC/ACM-2001, 2001.

[72] R. Lysecky, F. Vahid, T. Givargis, and R. Patel. Pre-fetching for Improved Core Interfacing. ISSS-1999, 1999.

[73] P. Magarshack and P.G. Paulin. System-on-Chip Beyond the Nanometer Wall. 40th Design Automation Conference (DAC-2003), USA, 2003.

[74] I. Mahgoub, A. Asaduzzaman, and M. Yousif. Evaluation of memory latency in cluster-based cache-coherent multiprocessor systems with different interconnection topologies. Computers & Electrical Engineering, Volume: 26, Issue: 3-4, pages 207-220, 2000.

[75] K. Malkowski, G. Link, P. Raghavan, and M.J. Irwin. Load Miss Prediction - Exploiting Power Performance Trade-offs. IEEE International Parallel and Distributed Processing Symposium (IPDPS-2007), pages 1-8, 2007.

[76] A. Mart, X. Molero, et al. Combined Intrinsic-Extrinsic Cache Analysis for Preemptive Real-Time Systems. Proceedings of 25th IFAC Workshop on Real-Time Programming, pages 49-55, 2000.

[77] A. Maxiaguine, S. Kunzli, and L. Thiele. Workload Characterization Model for Tasks with Variable Execution Demand. Proceedings of Design Automation and Test in Europe, Paris, France, 2004.

[78] A. Mihal and K. Keutzer. Mapping Concurrent Applications onto Architectural Platforms; A. Jantsch, H. Tenhunen (eds.), pages 39-59, Kluwer Academic Publishers, 2003.

[79] S. Mohanty and V.K. Prasanna. Rapid System-Level Performance Evaluation and Optimization for Application Mapping onto SOC Architectures. http://citeseer.ist.psu.edu/mohanty02rapid.html, 2002

[80] S. Mohanty, V.K. Prasanna. Design of High-Performance Embedded System using Model Integrated Computing; 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS-2004), 2004.

[81] S. Mohanty, V.K. Prasanna, S. Neema, and J. Davis. Rapid Design Space Exploration of Heterogeneous Embedded Systems using Symbolic Search and Multi-Granular Simulation. LCTES-2002/SCOPES-2002, Germany, 2002.

[82] A.M. Molnos, M.J.M. Heijligers, et al. Data Cache Optimization in Multimedia Applications. Proceedings of the 14th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC'03), Veldhoven, The Netherlands, pages 529-532, 2003.

[83] S.S. Mukherjee, S.V. Adve, T. Austin, J. Emer, and P.S. Magnusson: Performance Simulation Tools. IEEE Computer, 2002.

[84] M.D. Nava, R. Wilson, et al. An Open Platform for Developing Multiprocessor SOCs; IEEE Computer Society, pages 60–67, 2005.

[85] A. Naz, K. Kavi, M. Rezaei, and W. Li. Making a Case for Split Data Caches for Embedded Applications. ACM SIGARCH Computer Architecture News, Vol. 34, Issue 1, pages 19-26, 2006.

[86] W. Nebel. System-Level Power Optimization. IEEE/DSD, 2004.

[87] K.J. Nesbit and J.E. Smith. Data Cache Pre-fetching Using a Global History Buffer. IEEE-2005, pages 90-97, 2005.

[88] E. Nilsson and J. Oberg. Reducing Power and Latency in 2-D Mesh NoCs using Globally Pseudochronous Locally Synchronous Clocking. CODES+ISSS, pages 176-181, 2004.

[89] L. Nosetti, C. Solomon and E. Macii. Analysis of Memory Accesses in Embedded Systems. IEEE-1999, pages 1783-1786, 1999.

[90] P.R. Panda, F. Catthoor, P.G. Kjeldsberg, et al. Data and Memory Optimization Techniques for Embedded Systems. ACM Transactions on Design Automation of Electronic Systems, Vol. 8, No. 2, pages 149-206, 2001.

[91] J. M. Paul, C.P. Andrews, A.S. Cassidy, and D.E. Thomas. System-Level Modeling of a Network Switch SOC; ACM ISSS'02, Kyota, Japan, pages 62–67, 2002.

[92] J.M. Paul, D.E. Thomas, and A.S. Cassidy. High-level modeling and simulation of single-chip programmable heterogeneous multiprocessors. ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 10, Issue 3, pages 431-461, 2005.

[93] N. Pazos, W. Brunnbauer, J. Foag, and T. Wild. System level performance estimation of multi-processing, multi-threading SoC architectures for networking applications. SystemC: methodologies and applications book contents, pages 157-190, 2003.

[94] R. Pendse and H. Ksitta. Selective Pre-fetching: Pre-fetching when only required. IEEE-1999, pages 866-869, 1999.

[95] R. Pendse and R. Bhagavathula. Performance of LRU Block Replacement Algorithm with Pre-fetching. IEEE-1999, pages 86-89, 1999.

[96] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman. Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-Chip. CASES-2003, 2003.

[97] I. Puaut. Cache Analysis Vs Static Cache Locking for Schedulability Analysis in Multitasking Real-Time Systems. http://citeseer.ist.psu.edu/534615.html, 2006.

[98] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard realtime systems: a quantitative comparison. Design, Automation & Test in Europe Conference & Exhibition (DATE-2007), pages 1-6, 2007.

[99] I. Puaut and D. Decotigny. Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. IEEE, 2002.

[100] S. Rader, J. Corleto-Mena, N. Marshall, et al. Mobile Extreme Convergence: A Streamlined Architecture to Deliver Mass-Market Converged Mobile Devices. Freescale Semiconductor, 2005.

[101] S. Rajagopal, J.R. Cavallaro, and S. Rixner. Design space exploration for real-time embedded stream processors. IEEE, 2004.

[102] R.M. Ramanathan. Intel Multi-Core Processors: Making the Move to Quad-Core and Beyond. White Paper, Intel, 2006.

[103] P. Ranganathan, S. Adve, and N.P. Jouppi. Reconfigurable Caches and their Application to Media Processing. ISCA/ ACM, pages 214–224, Vancouver, Canada, 2000.

[104] P. Reed, M. Alexander, et al (Motorola). A 66-MHz Configurable Secondary Cache Controller with Primary Cache Copy-back Support. IEEE-1992, pages 16-17, 1992.

[105] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing Predictability of Cache Replacement Policies; Real-Time Systems Journal, Vol. 37, No. 2, pages 99-122, http://www.springerlink.com/content/l128713825873h30/, 2007.

[106] I. Richardson. H.264 / MPEG4 Part 10: Overview, in www.vcodex.com, 2002.

[107] A.E. Rizzoli. A Collection of Modeling and Simulation Resources on the Internet. IDSIA, Switzerland. http://www.idsia.ch/~andrea/sim/simindex.html, 2007.

[108] J. Robertson and K. Gala. Instruction and Data Cache Locking on the e300 Processor Core. Freescale Semiconductor, 2006.

[109] V. Romanchenko. Evaluation of the multi-core processor architecture Intel core: Conroe, Kentsfield…. Digital-Daily.com, 2006.

[110] V. Romanchenko. Quad-Core Opteron: architecture and roadmaps. Digital-Daily.com, 2006.

[111] R. Sangireddy, H. Kim, and A.K. Somani. Low-Power High-Performance Adaptive Computing Architectures for Multimedia Processing. Proceedings of the 9th International Conference on High Performance Computing, pages 124-136, 2002.

[112] R. Schaphorst. Videoconferencing and Videotelephony – Techonology and Standards, in Artech House, Norwood, MA, 2nd ed. 1999.

[113] L. Schoeb and G. Darnell. Large Processor L2 Cache Sizes in Dell PowerEdge Servers. Dell, 1999.

[114] F. Sebek and J. Gustafsson. Determining the Worst Case Instruction Cache Miss-Ratio. http://citeseer.ist.psu.edu/cache/papers/cs/...

[115] M. Shalan and V.J. Mooney. A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip. ACM CASES-2000, San Jose, CA, 2000.

[116] M. Shalan and V.J. Mooney. Hardware Support for Real-Time Embedded Multiprocessor System-on-a-Chip Memory Management. CODES-2002, CO, USA, 2002.

[117] A.L. Shimpi. Understanding the Cell Microprocessor. CPU & Chipset, 2005.

[118] T. Simunic and S. Boyd. Managing Power Consumption in Networks on Chips. IEEE, 2002.

[119] N.T. Slingerland and A.J. Smith. Cache Performance for Multimedia Applications. UCB, 2000.

[120] N.T. Slingerland and A.J. Smith. Design and Characterization of Berkeley Multimedia Workload. Multimedia Systems, Springer-Verlag, pages 315–327, 2002.

[121] P. Soderquist and M. Leeser. Optimizing the Data Cache Performance of a Software MPEG-2 Video Decoder. ACM Multimedia 97 – Electronic Proceedings, Seattle, WA, 1997.

[122] M. Soryani, M. Sharifi, and M.H. Rezvani. Performance Evaluation of Cache Memory Organizations in Embedded Systems. Fourth International Conference on Information Technology: New Generations (ITNG-2007), pages 1045-1050, 2007.

[123] J. Starner. Controlling Cache Behavior to Improve Predictability in Real-Time Systems. 2005.

[124] J. Stokes. Introducing the IBM/Sony/Toshiba Cell Processor – Part II: The Cell Architecture. http://arstechnica.com/articles/paedia/cpu/cell-2.ars, 2005.

[125] T. Suh, D. Blough, and H.S. Lee. Supporting Cache Coherence in Heterogeneous Multiprocessor Systems. Design, Automation and Test in Europe Conference and Exhibition Volume II (DATE-2004), 2004.

[126] T. Suh, D. Kim, and H.-H.S. Lee. Cache Coherence Support for Non-Shared Bus Architecture on Heterogeneous MPSOCs. Proceedings of the 42nd annual conference on Design automation. CA, USA, pages 553-558, 2005.

[127] P.F. Sweeney, M. Hauswirth, et al. Understanding Performance of MultiCore Systems using Trace-based Visualization. STMCS-2006, Manhattan, NY, 2006.

[128] E. Tamura, F. Rodriguez, J.V. Busquets-Mataix, and A.M. Campoy. High Performance Memory Architectures with Dynamic Locking Cache for Real-Time Systems. Proceedings of the 16th Euromicro Conference on Real-Time Systems, pages 1-4, Italy, 2004.

[129] E. Tamura, J.V. Busquets-Mataix, J.J.S. Martin, and A.M. Campoy. A Comparison of Three Genetic Algorithms for Locking-Cache Contents Selection in Real-Time Systems. Proceedings of the International Conference in Coimbra, Portugal, 2005.

[130] T. Tarui, T. Nakagawa, N. Ido, M. Asaie, and M. Sugie. Evaluation of the lock mechanism in a snooping cache. ACM Proceedings of the 6th international conference on Supercomputing, 1992.

[131] L. Thiele and R. Wilhelm. Design for Timing Predictability. Real-Time Systems, Vol. 28, Issue 2-3, pages 157-177, 2004.

[132] T. Tian. Intel Corp. Effective Use of the Shared Cache in Multi-core Architectures. Dr. Dobb's Portal. URL:http://www.ddj.com/embedded/196902836, 2007.

[133] K. Tiensyrja and A. Jantsch. NOCARC - Network On Chip Architectures. EXSITE Workshop, Sweden, 2001.

[134] G. Torres. Inside Pentium 4 Architecture. http://www.hardwaresecrets.com/printpage/235/1, 2005.

[135] R.A. Uhlig and T.N. Mudge. Trace-driven Memory Simulation: A Survey. ACM Computing Surveys, 1997.

[136] O.S. Unsal, R. Ashok, I. Koren, C.M. Krishna, and C.A. Moritz. Cool-Cache for Hot Multimedia. Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture, pages 274–283, 2001.

[137] A. Vance. Cell processor goes commando. Mountain View. http://www.theregister.co.uk/2006/01/22/cell_mecury_army/, 2006.

[138] G. Varatkar and R. Marculescu. Traffic Analysis for On-chip Networks Design of Multimedia Applications. ACM DAC-2002, New Orleans, Louisiana, 2002.

[139] X. Vera and B. Lisper. Data Cache Locking for Higher Program Predictability. SIGMETRICS-2003, CA, 2003.

[140] X. Vera, B. Lisper, and J. Xue. Data Caches in Multitasking Hard Real-Time Systems. IEEE RTSS-2003, 2003.

[141] F.J. Villa, M.E. Acacio, and J.M. Garcia. On the Evaluation of Dense Chip-Multiprocessor Architectures. IEEE, 2006.

[142] M. Vorbach, and J. Becker. Reconfigurable Processor Architectures for Mobile Phones. Proceedings of the 17th International Symposium on Parallel and Distributed Processing, 2003.

[143] R. Wilhelm. Timing Analysis and Timing Predictability; FMCO-2004, Vol. 3657 of LNCS, pages 317-323, 2005.

[144] R. Wilhelm and L. Thiele. Timing Predictability — a Must for Avionics Systems. 2006.

[145] R. Wilhelm, J. Engblom, S. Thesing, and D. Whalley. The Determination of Worst-Case Execution Times. ARTIST, 2003.

[146] W. Wolf. Multimedia Applications of Multiprocessor Systems-on-Chips. IEEE, 2005.

[147] W. Wolf. The Future of Multiprocessor Systems-on-Chips. DAC-2004, pages 681–685, San Diego, CA, 2004.

[148] W. Wolf and M. Kandemir. Memory System Optimization of Embedded Software. Proceedings of the IEEE, Vol. 91, No. 1, pages 165-182, 2003.

[149] Z. Wu, M. Tokumitsu, and T. Hatanaka. The Development of MPEG4-AVC/H.264, the Next Generation Moving Picture Coding Technology, in Issue 200 Vol.71 No.4, 2004.

[150] Q. Xu and P.J. Teller. Unified vs. split TLBs and caches in shared-memory MP systems. 9th International Parallel Processing Symposium (IPPS-1995), page 398, 1995.

[151] J. Xu, W. Wolf, J. Henkel, and S. Chakradhar. A Methodology for Design, Modeling, and Analysis of Networks-on-Chip. IEEE, pages 1778-1781, 2005.

[152] J.J. Yi, L. Eeckhout, D.J. Lilja, B. Calder, L.K. John, and J.E. Smith. The Future of Simulation: A Field of Dreams? The IEEE Computer Society, pages 22-29, 2006.

[153] C. Zhang and F. Vahid. Using a Victim Buffer in an Application-Specific Memory Hierarchy. Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE-2004), 2004.

[154] C. Zhang, F. Vahid, and R. Lysecky. A Self-Tuning Cache Architecture for Embedded Systems. ACM Transactions on Embedded Computing Systems, Vol. 3, No. 2, pages 407-425, 2004.

[155] Y. Zheng, B.T. Davis, and M. Jordan. Performance Evaluation of Exclusive Cache Hierarchies. IEEE-2004, pages 89-96, 2004.

[156] Application Note 32: The ARMulator, ARM Limited, ARM DAI 0032F. 2003. http://www.arm.com/support/ARMulator.html

[157] Basics of Cell Architecture; Sony Computer Entertainment Inc. 2006. http://multimedia.cx/linux-20061110-docs/CellProgrammingTutorial/BasicsOfCellArchitecture.html

[158] Cache - Smart Computing Encyclopedia. 2009. http://www.smartcomputing.com/editorial/dictionary/detail.asp?guid=&searchtype=&DicID=16600&RefType=Encyclopedia

[159] Cachegrind – A cache profiler from Valgrind. 2009.
http://valgrind.kde.org/index.html

[160] Embedded System. From Wikipedia, the free encyclopedia, 2007.

[161] FFmpeg – A very fast video and audio converter. 2009.
http://ffmpeg.sourceforge.net/ffmpeg-doc.html#SEC1

[162] Heptane – A tree-based WCET analysis tool. 2007.
http://ralyx.inria.fr/2004/Raweb/aces/uid43.html

[163] Improving cache performance. UMBC. 2007.
http://www.csee.umbc.edu/help/architecture/611-5b.ps

[164] JM-RS (96) – H.264/AVC Reference Software. 2009.
http://iphome.hhi.de/suehring/tml/ download/

[165] MPEG Software Simulation Group: Mpeg2 Encoder and Decoder. 2009.
http://tns-www.lcs.mit.edu/manuals/mpeg2/index.html

[166] Multi-core (computing). 2009.
http://en.wikipedia.org/wiki/Xeon. http://en.wikipedia.org/wiki/Athlon

[167] NTNU Computer Architecture Research Group. 2007.
http://www.idi.ntnu.no/~dam/ncar/

[168] Reducing Memory Pressure for Chip-Multiprocessors using Cache Injection. leon:research:cache_injection [SSL Wiki], 2009.

[169] System-on-a-chip (SOC). 2009.
http://en.wikipedia.org/wiki/System-on-a-chip

[170] The Cache Memory: An Unpredictable Hardware Component in Single- and Multiprocessor Real-Time Systems. 2005.
http://www.idt.mdh.se/fsk/docs/cacheposter.pdf

[171] Video Technology Magazine. ITU H.263 Video Compression.2009.
http://www.h263l.com/

[172] VisualSim – A system-level simulator from Mirabilis Design, Inc. 2009.
http://www.mirabilisdesign.com/