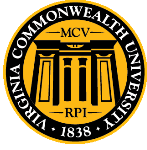

Virginia Commonwealth University
VCU Scholars Compass

Theses and Dissertations Graduate School

2013

STUDIES OF PERFORMANCE AND PREDICTABILITY OF CACHE-LOCKING

Matt Loach
Virginia Commonwealth University

Follow this and additional works at: https://scholarscompass.vcu.edu/etd

Part of the Engineering Commons

© The Author

Downloaded from
https://scholarscompass.vcu.edu/etd/497

This Thesis is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.

STUDIES OF PERFORMANCE AND PREDICTABILITY OF CACHE-LOCKING

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

by
Matthew R. Loach
B.S. in Computer Engineering, Southern Illinois University, 2011

Director: Dr. Wei Zhang
Associate Professor
Department of Electrical and Computer Engineering

Virginia Commonwealth University
Richmond, Virginia
May 2013

ACKNOWLEDGEMENTS

Thanks to Dr. Zhang for advising and supporting me and thanks to my family and friends that provided support and made this project possible.

Contents

ACKNOWLEDGEMENTS ii

LIST OF FIGURES vii

LIST OF TABLES xi

ABSTRACT xii

1 INTRODUCTION 1

1.1 BACKGROUND 1

1.1.1 Worse-Case-Execution Time 2

1.1.2 Architectural-Time-predictability Factor 3

1.1.3 Cache and Cache-Locking 3

1.1.4 Multi-core Architecture 5

1.1.5 Related Work 6

1.2 PURPOSE 8

1.2.1 Motivation 8

1.2.2 Thesis Research Overview 9

2 EVALUATING THE TIME PREDICTABILITY AND PERFORMANCE OF CACHE LOCKING 10
2.1 CHAPTER OVERVIEW . . . 10
2.2 MOTIVATION . . . 10
2.3 TIME PREDICTABILITY . . . 11
2.3.1 Dynamic Execution Time . . . 11
2.3.2 Static Prediction Time . . . 12
2.4 LOCKING METHODS . . . 12
2.4.1 Dynamic Locking . . . 12
2.4.2 Static Locking . . . 13
2.5 EVALUATION METHOD . . . 14
2.5.1 Dynamic Execution . . . 14
2.5.2 Locking . . . 15
2.5.3 Static Prediction . . . 16
2.5.4 Benchmarks . . . 18
2.6 RESULTS . . . 18
2.6.1 Dynamic Locking Threshold . . . 18
2.6.2 Benchmark Performance . . . 21
2.6.2.1 Cache Size of 32 Instructions . . . 21
2.6.2.2 Cache Size of 64 Instructions . . . 26
2.6.2.3 Cache Size of 128 Instructions . . . 27
2.6.2.4 Direct-Mapped Cache . . . 29

2.6.3 Time Predictability 31

2.7 CONCLUSIONS 36

2.7.1 Static Locking 36

2.7.2 Dynamic Locking 36

2.7.3 Architectural-Time-Predictability Factor 36

3 EXPLORING HYBRID CACHE-LOCKING TO BALANCE PERFORMANCE AND TIME PREDICTABILITY 38

3.1 CHAPTER OVERVIEW 38

3.2 MOTIVATION 39

3.3 HYBRID CACHE LOCKING 39

3.3.1 Locking Methods 40

3.3.2 Implementation 41

3.4 EVALUATION METHODOLOGY 41

3.4.1 Locking 42

3.4.2 Static Prediction 42

3.5 RESULTS 43

3.6 CONCLUSION 49

4 MULTI-CORE CACHE-LOCKING 51

4.1 CHAPTER OVERVIEW 51

4.2 MOTIVATION 52

4.2.1 Relevant work 53

4.3 LOCKING METHODS 53
4.3.1 Dynamic Locking 53
4.3.2 Static Locking 54
4.3.3 Effects on Time Predictability 55
4.4 IMPLEMENTATION 56
4.4.1 Thresholds 57
4.4.2 Benchmarks 58
4.5 RESULTS 59
4.5.1 L2 Cache 64
4.6 CONCLUSION 71
4.6.1 Cache Size 71
4.6.2 L2 Cache 71
4.6.3 General Purpose Processor 72

5 CONCLUSION **73**

5.1 CONCLUSION 73
5.1.1 Future work 75

Bibliography **76**

VITA **80**

List of Figures

1.1 Example of a Cache 4

2.1 Cache miss ratio with cache size of 32 instructions. 19

2.2 Cache miss ratio with cache size of 64 instructions. 20

2.3 Cache miss ratio with cache size of 128 instructions. 20

2.4 The average miss ratio with cache size of 32. 22

2.5 The average miss ratio with cache size of 64. 22

2.6 The average miss ratio with cache size of 128. 23

2.7 The overall average miss ratio of each dynamic locking threshold. . . . 23

2.8 Cache miss ratio for a cache size of 32 instructions. 24

2.9 Dynamic execution cycles for a cache size of 32 instructions. 25

2.10 Static prediction cycles for a cache size of 32 instructions. 25

2.11 Cache miss ratio for a cache size of 64 instructions. 27

2.12 Dynamic execution cycles for a cache size of 64 instructions. 28

2.13 Static prediction cycles for a cache size of 64 instructions. 28

2.14 Cache miss ratio for a cache size of 128 instructions. 29

2.15 Dynamic execution cycles for a cache size of 128 instructions. 30

2.16 Static prediction cycles for a cache size of 128 instructions. 30
2.17 Cache miss ratio for a direct mapped cache of 64 instructions. 31
2.18 Dynamic execution cycles for a direct mapped cache of 64 instructions. 32
2.19 Static prediction cycles for a direct mapped cache of 64 instructions. . . 32
2.20 Calculated ATF for a cache size of 32 instructions. 33
2.21 Calculated ATF for a cache size of 64 instructions. 34
2.22 Calculated ATF for a cache size of 128 instructions. 34
2.23 Calculated ATF for a cache size of 64 direct mapped instructions. 35

3.1 Hybrid cache-locking. 40
3.2 ATF as percent of lockable cache for the *edn* benchmark. 44
3.3 ATF as percent of lockable cache for the *adpcm* benchmark. 44
3.4 ATF as percent of lockable cache for the *cnt* benchmark. 45
3.5 ATF as percent of lockable cache for the *bsort* benchmark. 46
3.6 ATF as percent of lockable cache for the *fdct* benchmark. 47
3.7 ATF as percent of lockable cache for the *fir* benchmark. 47
3.8 ATF as percent of lockable cache for the *jfdcint* benchmark. 48

4.1 L1 Static-instruction locking, for size 16k, 32K, and 64K cache, showing the normalized cache miss ratio. 59
4.2 L1 Static-instruction locking showing the number of execution cycles for 16K, 23K and 64K caches, normalized to unlocked 16K cache. 60

4.3 L1 Dynamic-instruction locking, for size 16K, 32K, and 64K cache, normalized to 16K unlocked cache. 61

4.4 L1 Dynamic instruction locking, the number of execution cycles for size 16K, 32K and 64K cache, normalized to unlocked 16K cache. 61

4.5 Miss ratio for L1 static data locking, sizes 16K, 32K, and 64K, normalized to 16K unlocked cache's miss ratio. 62

4.6 L1 Static data locking, number of execution cycles for 16K, 23K and 64K caches, normalized to unlocked 16K cache. 62

4.7 Miss ratio for L1 instruction locking, size 32K cache, normalized to unlocked cache. 63

4.8 L1 instruction locking, cycles for size 32K cache, normalized to unlocked cache. 64

4.9 16K and 32K L1, 2M L2 dynamic cache-locking, instruction and data, miss ratio normalized to L1 size 16K unlocked cache. 64

4.10 16K and 32K L1, 2M L2 dynamic cache-locking, instruction and data, execution cycles, normalized to L1 size 16K unlocked cache. 65

4.11 32K L1, 1M and 2M L2 dynamic cache locking, instruction and data, miss ratio normalized to L2 size 1M unlocked cache. 65

4.12 32K L1, 1M and 2M L2 dynamic cache locking, instruction and data, execution cycles, normalized to L2 size 1M unlocked cache. 66

4.13 32K L1, 2M L2 dynamic cache-locking, comparing instruction, data and both instruction and data locking, miss ratio, normalized to unlocked cache. 67

4.14 32K L1, 2M L2 dynamic cache-locking, comparing instruction, data and both instruction and data locking, execution cycles, normalized to unlocked cache. 67

4.15 16K L1 and 32K L1, 2M L2, static cache-locking, miss ratio normalized to L1 size 16K, L2 size 2M unlocked cache. 68

4.16 16K L1 and 32K L1, 2M L2, static cache-locking, execution cycles, normalized to L1 size 16K, L2 size 2M unlocked cache. 69

4.17 32K L1, 1M-2M L2, static cache locking, miss ratio, normalized to L2 size 1M unlocked cache. 69

4.18 32K L1 1M-2M L2 Static cache-locking, execution cycles, normalized to L2 size 1M unlocked cache. 70

List of Tables

2.1 SimpleScalar Configuration 14

2.2 Cache Configurations 15

2.3 Benchmark Information 18

4.1 Simulator Configuration L1 Locking 57

4.2 Simulator Configuration L2 Locking 57

4.3 Simulator Configuration L2 Locking Cont. 57

4.4 Benchmarks 58

ABSTRACT

STUDIES OF PERFORMANCE AND PREDICTABILITY OF CACHE-LOCKING

By Matthew R. Loach, M.S.

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

Virginia Commonwealth University, 2013

Director: Dr. Wei Zhang,
Associate Professor, Department of Electrical and Computer Engineering

Today's real-time systems need to be faster and more powerful than ever before. Caches are an architectural feature that helps solve this problem. Caches however are unpredictable and do not improve the worst case execution time. This work studies the effects of cache-locking on performance and time predictability. Two locking methods were evaluated: a dynamic locking method and a static locking method. The performance of single and multi-core processors and multiple levels of caches were studied. The time predictability of the single core system was studied and the cost of the time predictability was determined for each locking method. Cache-locking in the Level 2 cache had the best performance and the static-locking method had the highest predictability.

Chapter 1 INTRODUCTION

1.1 BACKGROUND

A real-time system is a computer processing system that takes input generated from outside of it and processes it by a definite, specific time deadline. Real-time systems are used everywhere in our daily life from safety and piloting systems on airplanes to entertainment devices. The difference between normal general purpose systems and real-time systems is that, in real-time systems, the work done is divided into tasks and each task has a specific deadline by which it needs to be completed. Thus it is crucial to determine that the real-time system will meet its task deadline. Depending on the tasks, it may be more important that the deadline is predictable (time-predictable) than how fast it completes the tasks (performance). Figuring out whether it will meet the deadline involves estimating the worst-case time. The worst-case execution time (WCET)is defined as the time it will take the task to complete.

Real-time systems are split into two categories; hard and soft real-time systems. The difference between them is that in hard real-time systems, missing a deadline means a catastrophic failure. Soft real-time systems are allowed to sometimes miss deadlines.

General purpose processors are not usually used in real-time systems even though they are very fast and powerful. The technology that makes them so fast is the reason why they are not used; technology such as caches and branch prediction are inherently unpredictable. There is a fundamental difference between the design of a general purpose processor and a real-time system processor. The general purpose processor focuses on the average case and implements features that try to improve it as much as possible, even if the features degrade the worst case. Real-time-processor design focuses on improving the worst-case execution, which makes many of the advanced features in processors useless in their general purpose configuration. The Texas Instruments C2000 Microcontrollers or the Sitara ARM Cortex-A8 Microprocessors are examples of normal processors for real-time systems. Microprocessors like the C2000 are very simple compared to general purpose processors like those of Intel.

1.1.1 Worse-Case-Execution Time

One method of evaluating a real-time system is by measuring the average delay, where the less delay, the more predictable it is. Another method that is also used for figuring out deadlines is estimating the WCET. The WCET is the largest amount of time a task can possibly take to complete. For unpredictable features like cache and branch prediction there is some research in creating a tighter estimation than WCET, but the calculations become very complex and, in many cases, computationally unfeasible. [1]

1.1.2 Architectural-Time-predictability Factor

Architectural-Time-predictability Factor (ATF) [8] is a method of measuring the predictability of a certain architecture or architectural feature. The usual method of measuring the average delay is not a deterministic or even accurate method of measuring predictability. The ATF is discussed in more detail in later chapters.

1.1.3 Cache and Cache-Locking

To increase the speed (performance) of computer systems, a cache memory is used. The cache shortens the time it takes to access instructions. Cache stores instructions so that when the instructions are used again, they can be accessed faster. Data in the cache results in a cache hit which is faster than when the data is not in the cache; a cache miss. The time (latency) it takes to retrieve data from a miss of the cache takes longer. The cache, therefore, can speed up the performance of the processing system. Since instructions in the cache can be replaced by other instructions as the processor runs, some instructions can be removed that are later needed. This causes cache misses and results in unpredictability in the system which causes problems in real-time systems.

There has been much research on cache as a source of unpredictability in real-time systems. From this research, one of the solutions to cache unpredictability is a method called cache-locking or cache-partitioning. Normally a WCET estimation has to assume that data and instructions are not in the cache because the state of the cache

is only known at run time and estimation happens before. Cache-locking locks data into the cache so that the data is predictably going to be available in the cache. This way the WECT estimation does not have to assume that the data is in the main memory because it is locked into the cache, allowing for a better WCET. The downside to cache-locking is that the average case performance decreases significantly because the cache is not able to utilize the space of the locked data to put new data in. There has been much research focused on trying to improve the performance of cache-locking.

FIGURE 1.1: Example of a Cache

Figure 1.1 is an example configuration of a cache with the ability to lock cache lines or blocks into the cache. The tag is the a reference to the address of the cache block and the V column is called the valid bit and it signifies whether the cache block is ready of be read or is invalid and needs to be fetched from the memory. The L column is for cache locking, a 0 signifies that the block is unlocked and can be replaced by

the replacement method, while a 1 signifies that the blocks is locked and can not be replaced.

1.1.4 Multi-core Architecture

There are many intensive tasks, not only for the general purpose processors, but also for the real-time processors. Many of these tasks can not be completed fast enough by singe core processors, or not without high energy costs due to increasing the frequency of the processor. The solution is multi-core processors. They have more through-put and use less power. An example is computer vision. Cars available today are using computer vision for driving, but keeping track of objects and processing all the data is a very intensive task and it has to be done in real-time. The systems need to detect a collision with an object before it hits the car. To supply the computer power for these intensive tasks, general purpose multi-core processors could be the solution if their real-time unpredictability issues can be sorted out. General purpose multi-core processors are split into two general categories: the massively parallel and the slightly parallel tasks. General-purpose graphics processing units (GPGPUs) are examples of massively parallel hardware and today's multi-core CPUs deal with slightly parallel tasks that usually have less than 100 threads. Analysis of the WCET for general purpose multi-core processors is difficult because different threads may share a cache, but also have an advantage because different threads with the same instructions can share the instructions.

1.1.5 Related Work

Puaut [2] gives a comprehensive overview of the difficulties of WCET estimation of caches and how caches can be made more predictable by using static locking. The work defines *intra-task* interference where a task will replace its own cache blocks and *inter-task* interference where a task preempts and replaces cache blocks of a different task, causing a delay in reloading the cache after the preemption. This study is a work-in-progress.

Vera *et al.* [3] use cache-locking and compile-time analysis to decided what data blocks should be locked in the cache in order to decrease the reduction in performance. Although the authors were not able to eliminate all of the performance degradation, they were able to achieve an accurate estimation of the execution time. The method used in this paper is very similar to the static-locking method used in this thesis.

Two methods greedy of selecting cache blocks to lock are proposed in the paper of Puaut and Decotigny [4]. They use the memory access analysis of the worst case execution path. The first method minimizes the worst-case CPU utilization; the second one minimizes interference between tasks. Like the methods in this paper the methods in this thesis focus on maximizing performance by locking blocks that are going to be used the most.

Puaut and Arnaud [5] propose to statically lock cache contents offline per task in a hard real-time system where there can be no unpredictability. The researchers measure their success by using worst-case cache miss rates. This paper considers cache in an

extreme example, a hard real-time system, unlike this thesis which considers a general soft real time system that still has some unpredictable features.

In Asaduzzaman *et al.* [6], an embedded system is used to run real-time applications to evaluate the performance and predictability of instruction cache locking. It was found that the performance and time predictability were improved with up to 15% locking. After 15%, it only increased the time predictability. This paper is similar to Chapter 3 where the cache-locking varies and performance and time predictability are measured.

Liu *et al.* [7] deal with task assignment, cache partitioning and evaluation with WCET of a set of tasks. Much analysis is done on the combination of cache-locking, cache-partitioning and task assessment on multiple cores. Unlike this paper, this thesis focuses on only one task running at a time and although it may have multiple threads in the multi-core section of the thesis, each thread has its own core to run on. There are no other tasks that are preempting the running task. This paper does take a similar approach to evaluating the multi-core processors, although they use a system-on-chip. The authors are still investigating interference on a shared cache from each of the cores. This interference is relevant to Chapter 4 where a multi-core processor is evaluated using the cache-locking methods described in this thesis.

This thesis is unique in the way that it measures time predictability, other papers measure time predictability by measuring the delay of a task. In this work the measurement of time predictability is take a step farther by using the ATF to put an exact number to how time predictable an architectural feature is. Hybrid cache locking is another unique

feature that is explored in this thesis, as it give the ability to fine tune and control the cache locking methods.

1.2 PURPOSE

1.2.1 Motivation

Real-time-processing applications are not getting easier or smaller to compute and need more powerful processors to compute them within a reasonable time period. This means that unpredictable features like caches are needed to help speed up the processing of these applications. The problem is implementing these unpredictable features in a predictable way. Cache-locking is a popular method to make caches more predictable. It allows the contents of the cache to be statically known or know before the execution of the program. Cache-locking has a downside. If the wrong blocks are locked, then cache-locking decreases performance because none of the instruction or data blocks are located in the cache and have to be fetched from the next level of the memory hierarchy, which is slower. Before being able to use cache-locking effectively, the best locking method for that type of application needs to be known.

Today's processors are multi-core and it is becoming rare to see a single core processor. This is problematic for real-time systems because running multiple threads simultaneously introduces unpredictability because it is unknown statically or before execution what thread is going to run when. In terms of cache, it is unknown what thread is going to access what data when or on what core, making cache even more

unpredictable. Cache is not useful in a real-time system if it is unpredictable, because real-time systems have to use the worst-case execution to guarantee that the task is done by a certain time. The worst case of a cache is having to fetch from the main memory. If it is assumed that every piece of data and instructions is always fetched from the main memory, then the cache is not useful and is just using power. Even if the program usually finishes earlier, the system still waits until the deadline.

1.2.2 Thesis Research Overview

This thesis explores and evaluates two different locking methods. Chapter 2 explores the predictability of using the two locking methods on a simple superscalar processor and compares the performance and time predictability. The time predictability is measured by using the ATF. The next chapter, Chapter 3, expands on Chapter 2's work by evaluating a method that will allow for only portions of the cache to be locked in an effort to fine tune the cost of time predictability. Chapter 4 expands these locking methods to the complex general purpose multi-core processor equivalent of today's desktop processors and explores the performance impact these two cache-locking methods have.

Chapter 2 EVALUATING THE TIME PREDICTABILITY AND PERFORMANCE OF CACHE LOCKING

2.1 CHAPTER OVERVIEW

Cache-locking improves time predictability but this feature has not been quantitatively measured. This chapter describes a metric to calculate the time predictability of a superscalar processor using the dynamic execution time and the statically-predicted time. The metric shows which architectural features are predictable and the trade-off between predictability and performance.

2.2 MOTIVATION

There is an abundance of metrics to measure performance of different types of cache such as miss ratios and cycle counts, but there is not a way to measure the effect cache has on time predictability. Currently, there is no way to compare if one method of cache-locking is more predictable than another or how much performance the predictability costs. Also a metric is needed to compare trade-offs between one architecture and another and also between other architectural features besides the cache. The

metric this thesis uses could help, not only compare, but decide which cache-locking method is the best, for the given application based on the needed performance and predictability versus the complexity of the method.

2.3 TIME PREDICTABILITY

The present work uses a metric for measuring the time predictability of superscalar processors using a concept originally proposed in [8]. That work developed the Architectural-Time-Predictability Factor (ATF) shown in Equation 2.1 and evaluated it on a Very Long Instruction Word processor (VLIW). The metric is a way to measure architectural components that are inherently not predictable [8]. This extends the study by using the metric to show how instruction cache-locking on a superscalar processor affects the time predictability. The metric takes the dynamic execution time and divides it by the statically-predicted time. This results in the ATF of the given configuration. The ATF is measured by how close to one the result of the metric is, with one being 100

$$ATF = \frac{Dynamic(T)}{Static(T)} \tag{2.1}$$

2.3.1 Dynamic Execution Time

Dynamic execution time is the time it takes for the processor to complete the benchmark in either the physical processor or simulation. The dynamic execution is where the effect of the cache and branch prediction and other unpredictable features of the

processor are collected. In this work the dynamic execution time is in terms of the total number of execution cycles it takes the simulation to complete for a given benchmark.

2.3.2 Static Prediction Time

Static prediction time is computed at compile time and only information that is revealed to the compiler can be used as shown in Equation 2.2. Compilers already use some timing information on the architecture to create a schedule that utilizes all of the resources on the processor. Since the cache is a very dynamic feature, the compiler cannot know what will or will not be in the cache. This means that the prediction must predict memory access to be misses in the cache for the same reasons used for safe WCET estimates of cache-based memory hierarchies.

$$Static(T) = \sum_{n=\text{number of instructions}} (\text{execution time of instruction + memory latency} \\ + \text{cache latency + pipeline latency}) \quad (2.2)$$

2.4 LOCKING METHODS

2.4.1 Dynamic Locking

Dynamic locking happens at run time. If an instruction in the cache gets more fetches than the threshold for locking, it is locked into the cache until the program terminates.

The dynamic locking method does this for all instructions until the cache is full. This method does not do advanced analysis of the program, but rather operates on the fly with the simple concept that instructions that are used often should be locked into the cache. The downside of this method is that only a limited number of instructions can be locked into the cache and an instruction could be locked into the cache on its last use.

2.4.2 Static Locking

Static locking decisions are made by the compiler using the control flow graph (CFG). The compiler identifies loops and paths. Loops are set to be locked and all of the instructions inside the loop are also set to be locked unless they are inside an **if** statement. This is because an **if** statement creates multiple paths and the compiler does not know which one, if any, will be taken. The goal is to only lock instructions on paths that are certain to be executed and executed multiple times. When the program is run, the cache waits until an instruction is fetched before it checks the compiler's output to see if the instruction should be locked into the cache.

One benefit of waiting until the cache loads the instruction to lock it, is that, if the program takes a path that does not contain some of the instructions the compiler said to lock, then those instructions will not be sitting in the cache using up space. The downside is that every instruction still has a cache miss.

2.5 EVALUATION METHOD

2.5.1 Dynamic Execution

To calculate the dynamic execution information, the simulator, SimpleScalar [9], was used. SimpleScalar is the version used with Chronos [10]. The main difference between the original SimpleScalar and the Chronos version is that the Chronos version filters out the library calls from the statistics and it includes a filter that filters library calls from the cache as well. The only instructions loaded into the cache and counted are the user instructions. SimpleScalar was configured to run so that it would match the static prediction program that is explained later. The configurations for SimpleScalar are shown in Table 2.1 with the exception of the memory hierarchy. Branch prediction and speculation were turned off so that effects of the cache would not be influenced by other unpredictable features. The memory hierarchy consists of only one cache, a Level 1 Instruction cache with the configurations shown in Table 2.2.

TABLE 2.1: SimpleScalar Configuration

Fetch Queue Size	1
Branch Prediction	Perfect
Decode Width	1
Issue Width	5
Commit Width	5
Cache Hit Latency	1
Memory Latency	10

TABLE 2.2: Cache Configurations

Instructions	32	64	128	64
Block size (bytes)	8	8	8	8
Number of sets	8	16	32	64
Associativity	4	4	4	1

2.5.2 Locking

The dynamic locking method was implemented in SimpleScalar. A counter was added for each instruction to keep track of how many times it had been fetched. When the fetch count was greater or equal to the threshold, the instruction was locked into the cache if there was room.

Static locking is implemented by a separate program from the compiler but only uses information that the compiler would have. This program does the analysis of the CFG that it read from files outputted by Chronos. The program then decides which basic blocks are parts of loops and should be locked and which basic blocks are parts of **if** statements inside loops and should not be locked. This is done so that only blocks common to every path are locked and other blocks that may not be executed every loop iteration, are not locked in the cache and do not take up space that could be used for a more important block. The instructions that the program decides to lock are then written to a file and read by the simulator SimpleScalar and/or static prediction program. The simulator checks each instruction that is loaded into the cache and, if it is listed on the list of instructions to be locked by the compiler, it is locked in the cache for the rest of the program's execution.

2.5.3 Static Prediction

To create a static prediction, Algorithm 1 is used to calculate how many cycles it will take for the program to complete using only information that can be known statically or at compile time. The algorithm uses a list of instructions to be completed along with cache configuration information. The first thing the algorithm does is to find dependencies for all of the instructions. Then it starts the main loop and continues looping until all of the instructions have been executed. The first step that happens inside the main loop is the accounting for each of the pipeline stages. The pipeline stages are in reverse order to prevent race conditions.

Algorithm 1 Static prediction

Input: $Instructions(I), hit_threshold$
Output: $cycles$

1: **while** $\exists\ I \notin Commit\ stage$ **do**
2: if $writeback\ stage,\ calculate\ latency,\ I\ to\ next\ stage$
3: if $execution\ stage,\ calculate\ latency,\ I\ to\ next\ stage$
4: **if** $Fetcher\ stage\ is\ Empty$ **then**
5: **if** Static Locking & $(I\ is\ in\ a\ Loop\ and\ not\ in\ an\ if\ statement)$ **then**
6: $Lock \leftarrow true$
7: **else if** Dynamic Locking & $(I\ is\ in\ a\ (Loop \geq hit_threshold)$ **then**
8: $Lock \leftarrow true$
9: **end if**
10: **end if**
11: **if** Lock & $cache\ not\ full$ **then**
12: $fetch_time_remain \leftarrow cache_hit$
13: **else**
14: $fetch_time_remain \leftarrow main_mem$
15: **end if**
16: $cycle_count++$
17: **end while**
18: **return** $cycle_count$

The next task is to check the fetcher, which starts on line four, to see if it is empty and, if so, fetch the next instruction. This is where the cache-locking happens and the process is dependent on what type of locking method is being used. For static-locking, the instruction is analyzed to find out if it is a part of a loop. Then it is checked to make sure it is not an **if** statement or inside of one. Lastly, it checks if there is room in the cache's given configuration and, if all of these conditions pass, then the instruction is locked into the cache. Dynamic locking is the same, except to predict which instructions will be dynamically-locked, profiling information is used to get a weight for loops and **if** statements. If the instruction is to be locked, the fetch time is that of a cache hit and, if it is not going to be locked, then the remaining fetch time equals the latency of the main memory. Also not shown is the fact that the predictor knows which locking method is being used and will account for how many cache misses will occur before the instruction is locked. For example, with the dynamic locking method with a threshold of sixteen, the predictor assumes sixteen cache misses before the instruction is locked in the cache.

The next part of the algorithm checks for empty function units and finds ready instructions for them. Then it updates the remaining time for execution in each of the function units and if a function unit has completed processing an instruction, the instruction is marked as executed and the function unit is marked as empty. Lastly, the fetch unit is updated and the cycle count increased. When all instructions are committed, the algorithm outputs the number of cycles it took to complete.

TABLE 2.3: Benchmark Information

Benchmark	Dynamic Instructions	Static Instructions
adpcm	4310	863
bsort	2422	78
cnt	67746	128
edn	75352	574
expint	1990	126
fdct	1809	302
fir	261666	96
jfdcint	3914	380
ludcmp	5305	222
matmult	141222	144

2.5.4 Benchmarks

Table 2.3 shows the different benchmarks from The Mälardalen WCET research group [11] that were used and the number of dynamic and static instructions along with a short description. The dynamic instructions are for the path that was executed. The static instructions are of the whole program and are important because they give an idea of how many instructions would be used depending on the path the program takes.

2.6 RESULTS

2.6.1 Dynamic Locking Threshold

Figures 2.1, 2.2, and 2.3 show the cache miss ratios of the benchmarks for different thresholds of the dynamic locking method. The *matmult* and *edn* benchmarks show little change between hit thresholds because they both start out with loops that iterate more times than the largest hit threshold, causing identical results. The benchmarks,

adpcm and *ludcmp*, follow the expected trend where the higher the threshold, the better the miss ratio. The lower thresholds locked instructions that were not used as much as others later on in the program and therefore filled up the cache early on. The higher thresholds did not fill up the cache until later in the program allowing the cache to function normally.

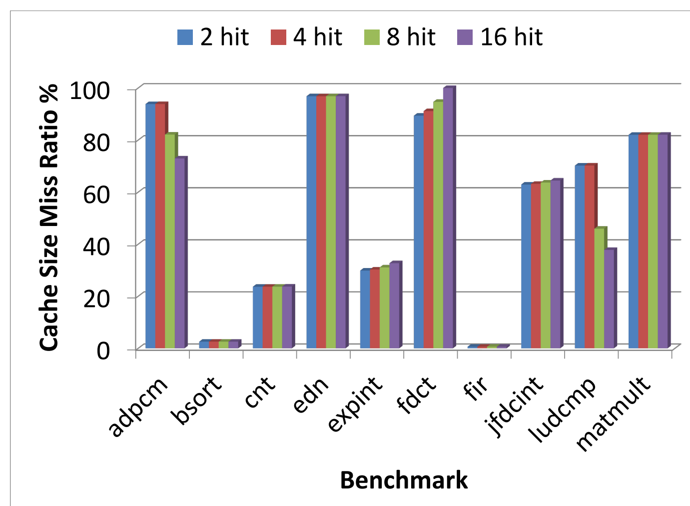

FIGURE 2.1: The cache miss ratios of dynamic locking for a cache size of 32 instructions with various benchmarks.

Three benchmarks had the opposite of the expected results: *expint*, *fdct* and *jfdcint*. These benchmarks are not made up of small loops with lots of iteration, but rather a large loop with few iterations, the worst case scenario for a normal cache. This is the one scenario that locking cache does better than non-locking cache because a normal cache would always be replacing blocks that it will need for the next iteration, resulting in all misses. The locked cache would take as many of those instructions as there is

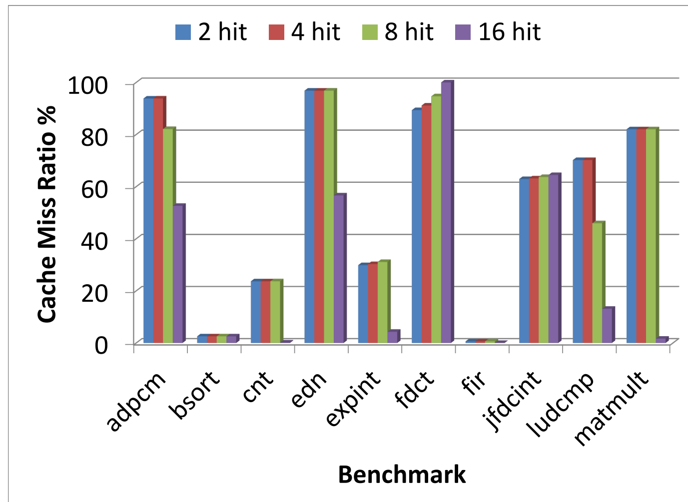

FIGURE 2.2: The cache miss ratios of dynamic locking for a cache size of 64 instructions with the various benchmarks.

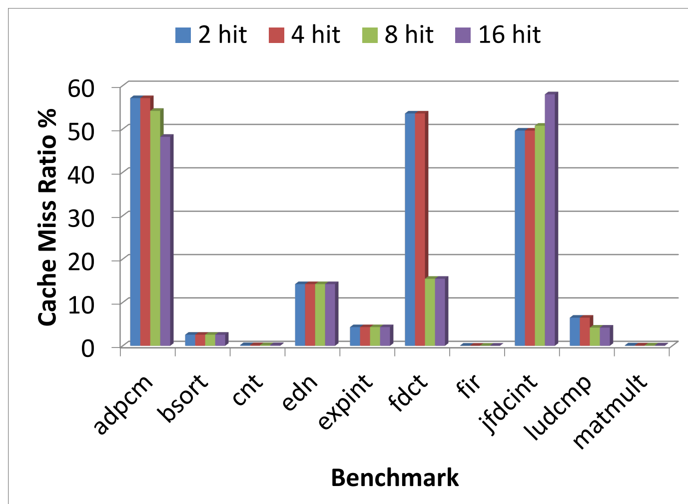

FIGURE 2.3: The cache miss ratios of dynamic locking for a cache size of 128 instructions with the various benchmarks.

room and lock them into cache, allowing for them to be hits every iteration. The lower thresholds did this while the higher thresholds either were not passed or locked only on the last couple of iterations. The benchmark, *fdct*, has a radical change with the increased cache size to 128 instructions because the active instructions can all fit in the cache where they could not before and since the large loops do not iterate more than the threshold, the cache does not fill up with locked blocks.

The average miss ratio for each cache size is shown in Figures 2.4, 2.5 and 2.6 and the overall average shown in Figure 2.7. Figure 2.7 shows that the higher thresholds have a better the miss ratio; 16 hits has 23% less cache misses than 8 hits. There are two performance reasons for the higher threshold being better. One is that the longer it takes to fill up the cache with locked blocks, the more the cache can act like a normal cache. The second reason is that having a higher threshold allows better blocks to be locked in most cases as explained above. These two reasons explain why having a higher threshold allows for better performance. The best performing dynamic locking threshold of 16 was used in the simulations in this chapter.

2.6.2 Benchmark Performance

2.6.2.1 Cache Size of 32 Instructions

Figures 2.8, 2.9 and 2.10 show the cache miss ratios, dynamic execution cycles and static prediction cycles of the benchmarks comparing the two different locking methods and non-locking, using a cache size of 32 instructions. Figure 2.8 shows the miss

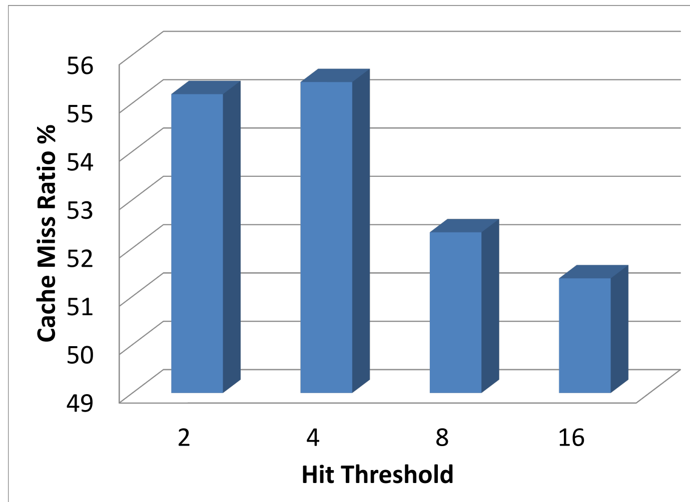

FIGURE 2.4: The average miss ratio of each dynamic locking threshold for a cache size of 32 instructions.

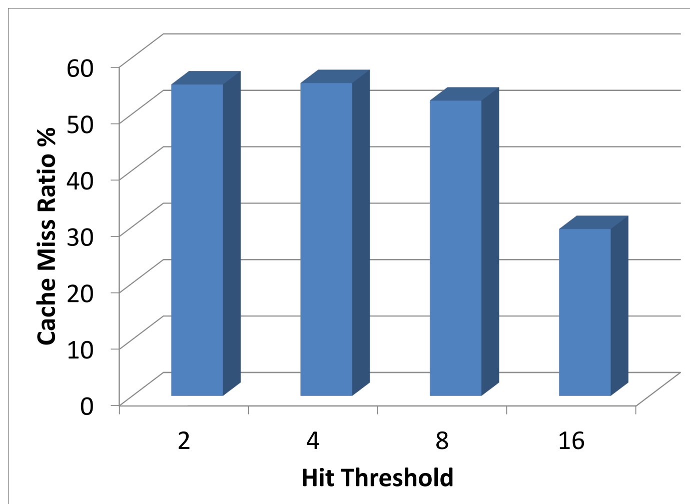

FIGURE 2.5: The average miss ratio of each dynamic locking threshold for a cache size of 64 instructions.

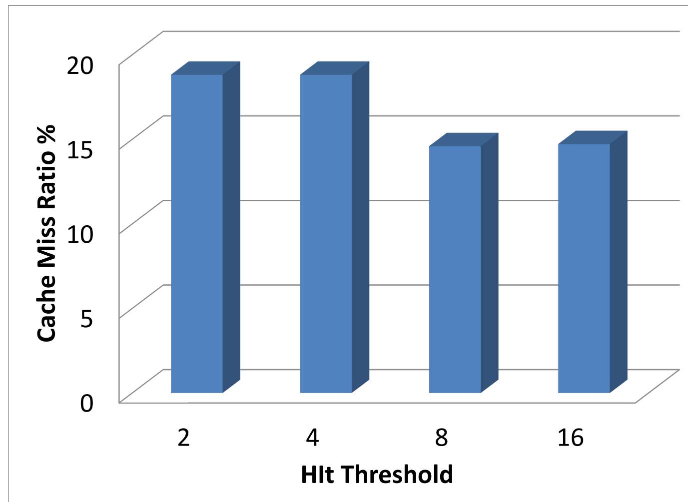

FIGURE 2.6: The average miss ratio of each dynamic locking threshold for a cache size of 128 instructions.

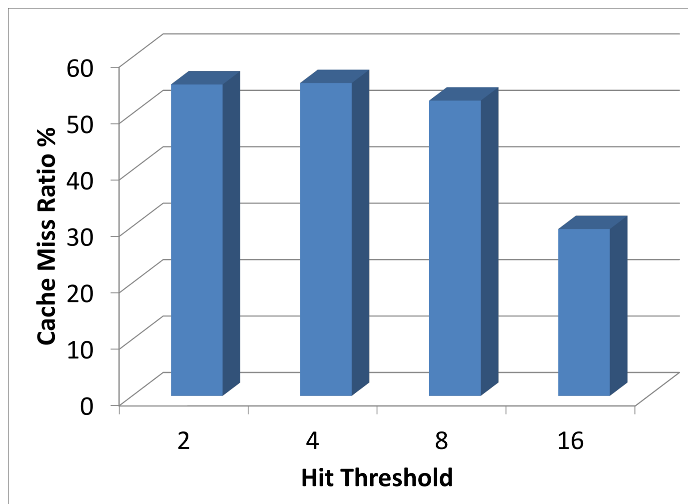

FIGURE 2.7: The overall average miss ratio of each dynamic locking threshold.

ratio and, as expected, the locking methods result in a higher miss ratio than non-locking. A few benchmarks like *cnt* and *ludcmp* have a slight anomaly; with the static method performing worse than the dynamic method. This is because the static method only locks instructions that are guaranteed to be used in a loop and most of the *cnt* instructions are conditional. Even though the static method selected instructions that were used every iteration, it left no room for the conditional instructions, resulting in all misses because they would be replaced before the next iteration. In contrast, the dynamic method was able to lock all but a couple instructions of the first loop into the cache and, even though all of the instructions in the following loops were misses, it still had less misses than the static locking method.

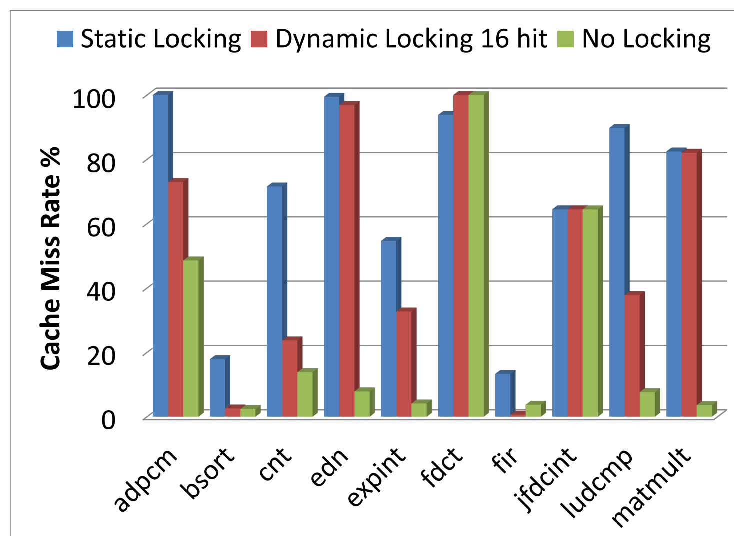

FIGURE 2.8: Cache miss ratio for a cache size of 32 instructions.

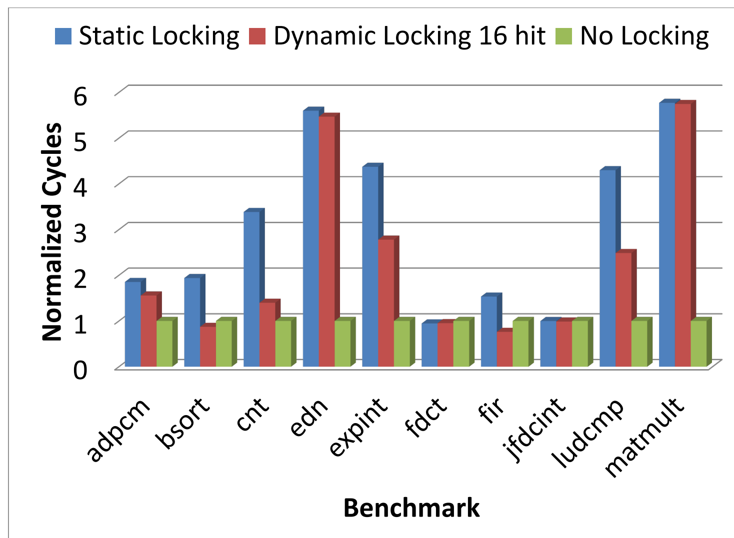

FIGURE 2.9: Dynamic execution cycles for a cache size of 32 instructions.

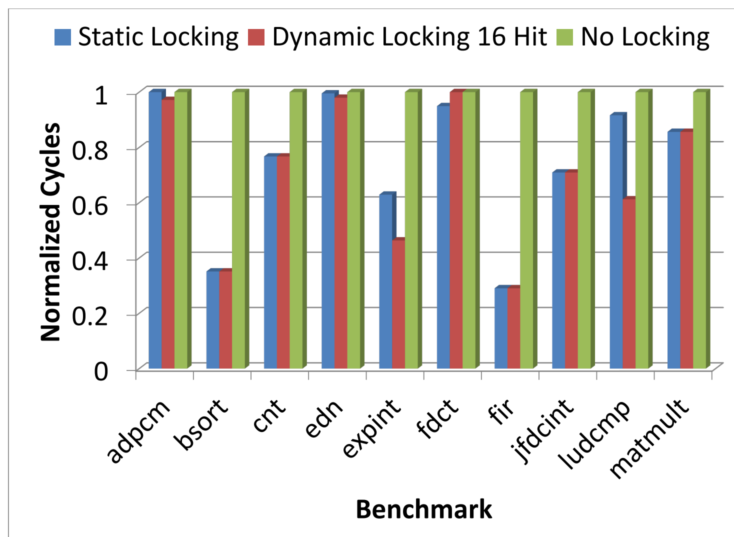

FIGURE 2.10: Static prediction cycles for a cache size of 32 instructions.

Another benchmark, *jfdcint*, also had odd cache miss ratio results. Both locking methods and the no-locking method performed similarly. The first loop of the program was loaded into the cache and all of the methods kept it in the cache. The locking methods locked the first loop into the cache, filling the cache and causing the second loop to be all misses. The second loop was too big for the cache and, therefore, also resulted in all misses for the no-locking method because the instructions would be replaced before they were used again.

Figure 2.9 reflects the cache miss ratio in figure 2.8, showing the effect that the cache has on the cycle count. For some benchmarks the effect of the cache looks nonexistent because Figure 2.9 is normalized to the no-locking method. Benchmarks like *fdct* and *jfdcint* do not appear to suffer from the cache because the no-locking method had the same or worse performance.

2.6.2.2 Cache Size of 64 Instructions

Figure 2.11, 2.12 and 2.13 show the same results as Figures 2.8, 2.9 and 2.10 respectively, but with a larger cache size of 64 instructions. Here we start seeing some of the benchmarks fitting into the cache and only having cold misses. The effect of the locking method on benchmarks like *bsort*, *cnt*, *expint* or *fir* is non-existent because there was not a need to remove any of the locked blocks. The benchmarks *ludcmp* and *matmult* are only minimally affected as they almost completely fit, but some of the instructions that were locked do get in the way, causing other instructions to be evicted or fetched directly from the main memory. Some of the benchmarks' predicted cycles

in Figure 2.13 also reflect the larger cache size because more locked blocks can be fit into the cache.

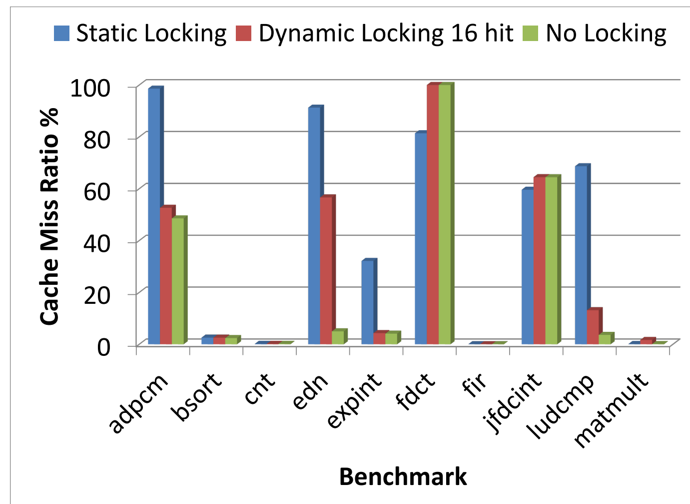

FIGURE 2.11: Cache miss ratio for a cache size of 64 instructions.

2.6.2.3 Cache Size of 128 Instructions

In a similar manner to the figures for cache sizes of 32 and 64, Figures 2.14, 2.15 and 2.16 show the results for a cache size of 128. With more space, the dynamic-locking method for *adpcm* and the static-locking method for *jfdcint* outperform or are equal to the no-locking method, unlike the small caches where they had worse performances. The benchmark *fdct* stands out because the static-locking method fills up the cache in the first loop, causing all the rest to miss, while the dynamic-locking does not end up

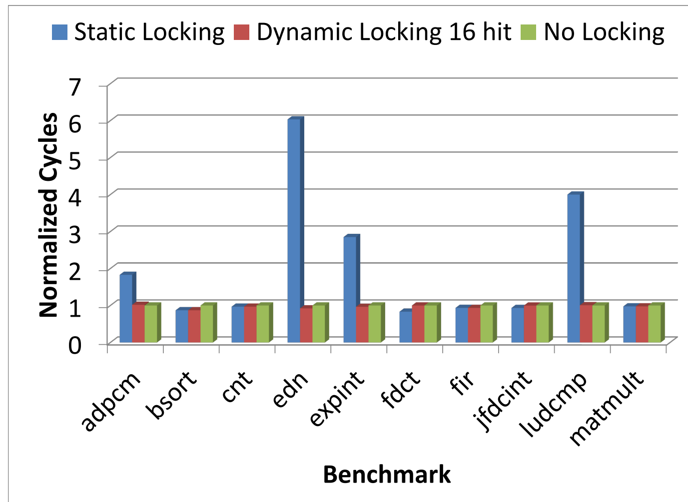

FIGURE 2.12: Dynamic execution cycles for a cache size of 64 instructions.

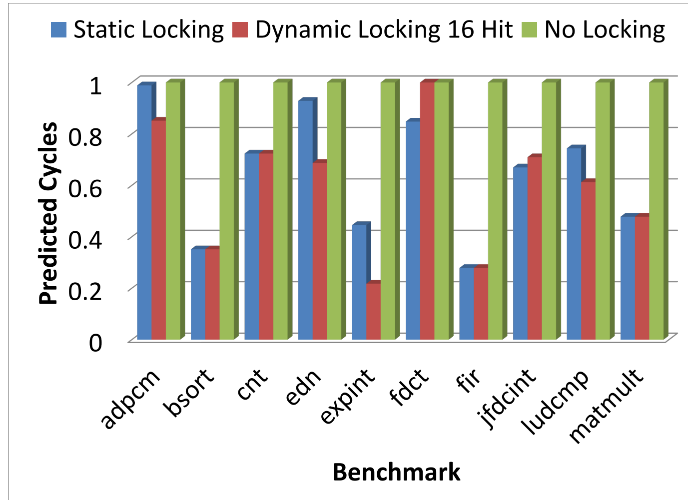

FIGURE 2.13: Static prediction cycles for a cache size of 64 instructions.

locking anything into cache because no instructions are used more than the threshold.

This causes the dynamic-locking method to have the same performance as the no-locking method.

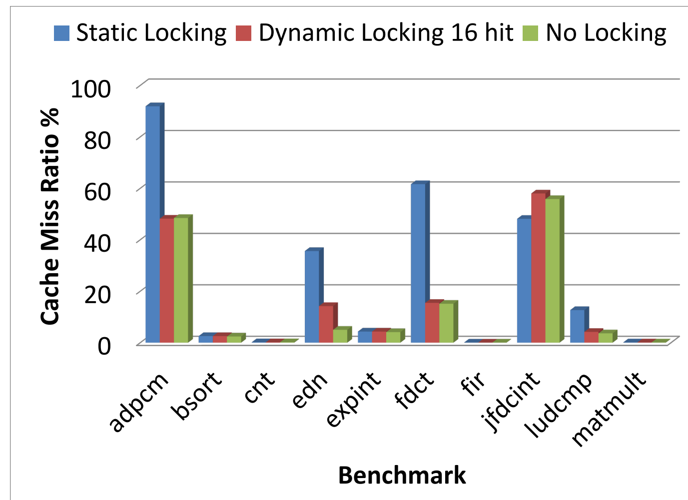

FIGURE 2.14: Cache miss ratio for a cache size of 128 instructions.

As cache size was increased to 64 and 128, more instructions could be locked, resulting in less cache misses. Some benchmarks were able to fit completely into the cache, like *bsort*, *cnt*, *expint* and *fir*, causing the locking methods to have no effect on their miss ratio.

2.6.2.4 Direct-Mapped Cache

Figures 2.17, 2.18 and 2.19 were included to show the benefit of associativity and how much it affects the locking methods. The *cnt* benchmark in Figure 2.17 shows

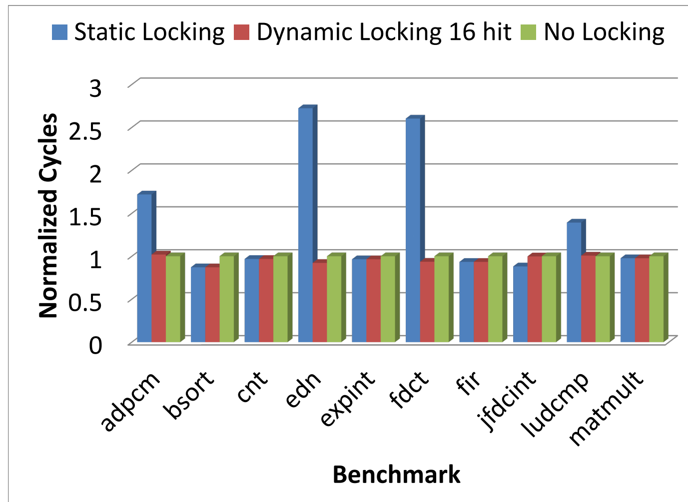

FIGURE 2.15: Dynamic execution cycles for a cache size of 128 instructions.

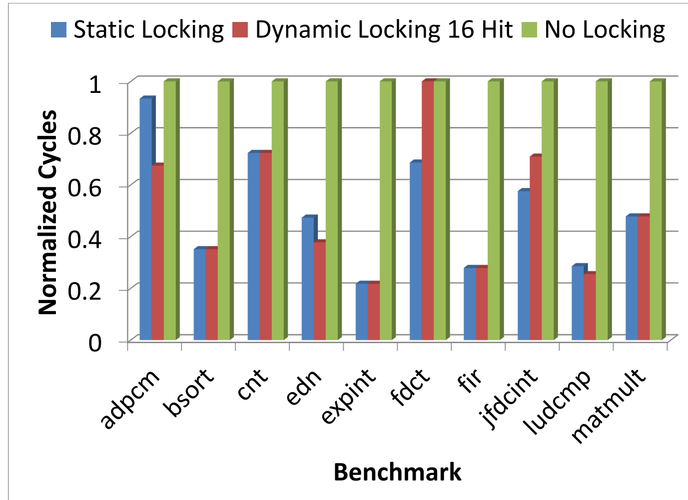

FIGURE 2.16: Static prediction cycles for a cache size of 128 instructions.

how the dynamic locking is more flexible in dealing with conflicts caused by the direct mapping compared to the static method that had a limited number of instructions that

were selected to be locked. Figure 2.18 matches the cache performance in Figure 2.17, reinforcing the effect that the direct mapping has when Figure 2.18 is compared to Figure 2.12. Comparing Figure 2.13 and Figure 2.19 shows that the associativity is also taken into account when the cycles are predicted.

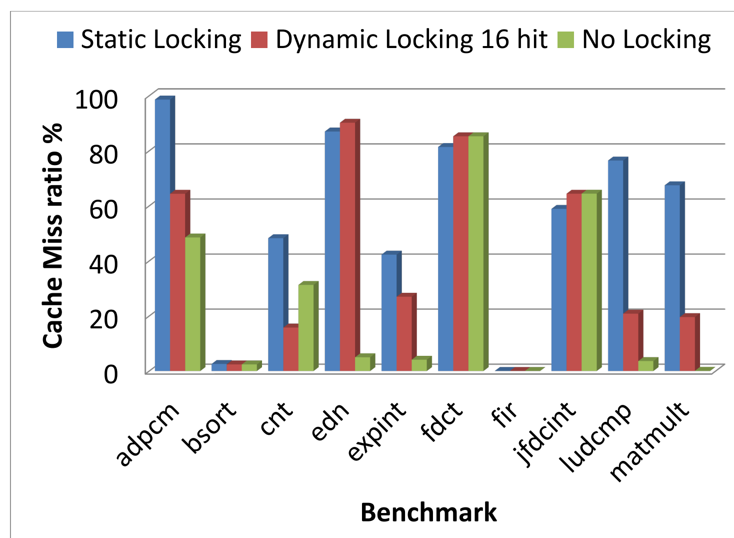

FIGURE 2.17: Cache miss ratio for a direct mapped cache of 64 instructions.

2.6.3 Time Predictability

Graphs of the calculated predictability for the different cache sizes are shown in Figures 2.20, 2.21 and 2.22. It was already well known [12] that the regular no-locking cache was very unpredictable and this metric gives it a number of how unpredictable it is for various different cache configurations. These graphs also show how time-predictable

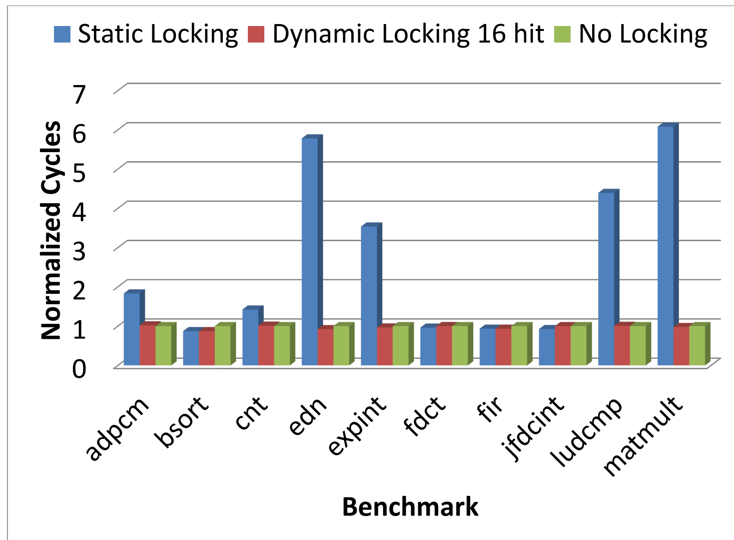

FIGURE 2.18: Dynamic execution cycles for a direct mapped cache of 64 instructions.

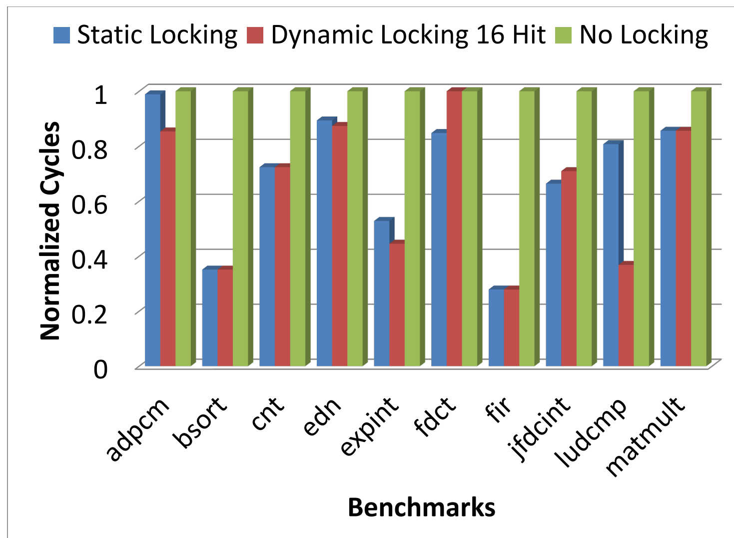

FIGURE 2.19: Static prediction cycles for a direct mapped cache of 64 instructions.

each cache-locking method is so that the methods can be compared to each other and regular no-locking cache by using the ATF. Also it is observed that the ATF decreases as the cache size increases, due to the fact that the predictor assumes cache misses for non-locked instructions and the cache may not get filled up with locked instructions.

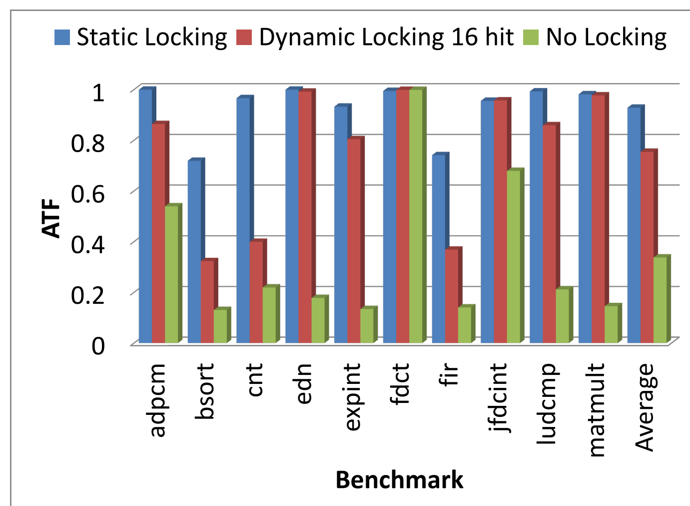

FIGURE 2.20: Calculated ATF for a cache size of 32 instructions.

The *edn* benchmark is an interesting example for cache-locking versus no cache-locking. As seen in Figure 2.9, *edn*'s dynamic execution cycles are so high because of its cache miss as shown in the Figure 2.8. Figure 2.9 shows that more than 90% of the cache accesses are missed because the cache was full of locked blocks. This benchmark shows the downside to cache-locking; the cache can be filled with locked instructions that are only used in the first half of the dynamic execution. The dynamic-locking method does better than the static because it chooses better blocks to lock into the cache, hence the lower miss ratio, but it still fills up the cache, preventing

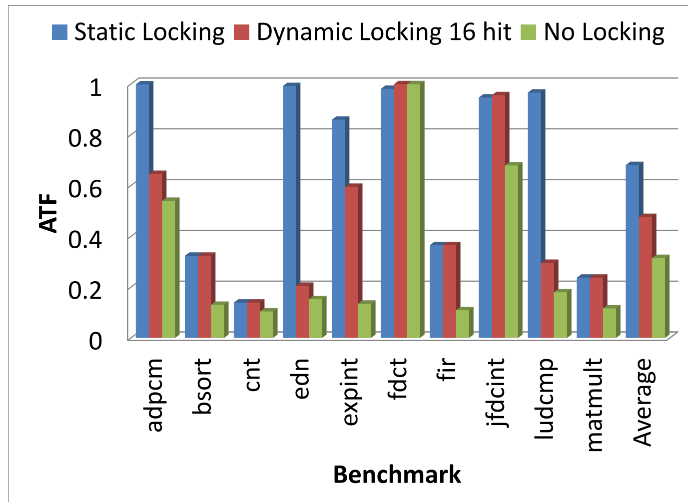

FIGURE 2.21: Calculated ATF for a cache size of 64 instructions.

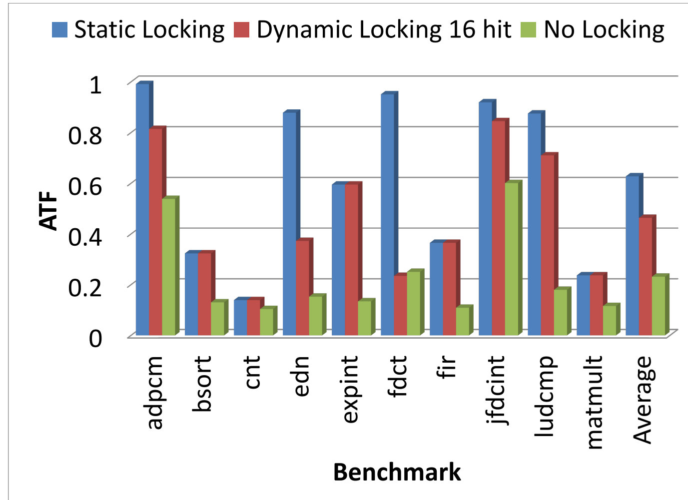

FIGURE 2.22: Calculated ATF for a cache size of 128 instructions.

instructions later in the program from being cached. The cache misses did increase the ATF because the predictor assumes cache misses and, as the cache got larger

and there were less cache misses, the ATF decreased. In the larger cache sizes the dynamic-locking method benefits from having more room more than the static method. The lower cache miss rate in the larger cache sizes result in lower predictability, but the predictability does not degrade proportionally to the cache miss rate.

The average ATF of the direct-mapped cache in Figure 2.23 is better than the associative cache in Figure 2.21 and the static-locking method surpasses the dynamic locking in time predictability. The ATF of the locking methods in the *matmult* benchmark increases, with the static-locking method dramatically increasing to 0.8 while the dynamic method only increases to 0.3. On average the static-locking method, even with its mapping conflicts due to being direct-mapped, still does better than the dynamic-locking which is more flexible.

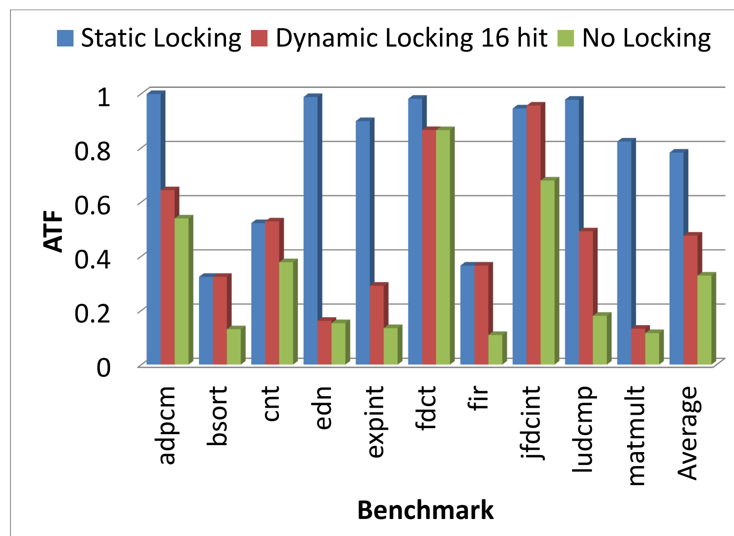

FIGURE 2.23: Calculated ATF for a cache size of 64 direct mapped instructions.

2.7 CONCLUSIONS

2.7.1 Static Locking

The static-locking method is the most predictable in the small cache size with an average ATF above 0.9. It does well because it was able to fill the cache with instructions that it choose to lock. However, for some benchmarks in the larger caches the static method was not able to select enough instructions to fill the cache, causing the lower ATF for some benchmarks.

2.7.2 Dynamic Locking

The dynamic-locking method was able to adapt to nearly any cache configuration with an average ATF between 0.73 and 0.47. Although adaptable is an adjective that is used in real-time systems, the dynamic-locking method requires no configuration or analysis to decide what to lock compared to the static-locking method and achieves a worst case average ATF of 0.47.

2.7.3 Architectural-Time-Predictability Factor

The metric presented in this chapter gives a number to the predictability of a cache and the different possible configurations. The results not only show the difference between the ATF and performance, but also show where they overlap. In general it can be inferred that a large cache is not as predictable as a small cache, but this is not quite

correct. It is not the cache size that causes unpredictability, but whether the locking method can utilize all of it. When all the cache is utilized, that increases the time predictability and the performance, but when not all the cache is used, the ATF suffers because the prediction does not know what was in the cache that was not locked. The smaller caches were more predictable because they could be filled easier with locked instructions. The associativity appears to cause some uncertainty and, while direct-mapping is more time-predictable, it takes a performance hit. The metric puts a number to what was only speculation before and will allow for intelligent architectural decisions when choosing instruction cache features.

Chapter 3 EXPLORING HYBRID CACHE-LOCKING TO BALANCE PERFORMANCE AND TIME PREDICTABILITY

3.1 CHAPTER OVERVIEW

This chapter explores hybrid cache-locking, a method that allows a fraction of cache lines to be locked; while the remaining cache lines remain unlocked. With hybrid cache-locking, performance and time predictability can be adjusted and tuned as compared to the two extremes, i.e., the regular cache without locking and the traditional cache-locking that can potentially lock all cache lines. Our evaluation indicates that hybrid cache-locking can often make better trade-offs between performance and time predictability, or provide more options as compared to either pure caching or pure locking. In some cases we observe that better performance can be achieved by locking only part of the cache.

3.2 MOTIVATION

Time predictability is crucial for real-time systems. Cache memories, while useful for performance, are harmful to time predictability. Cache-locking is a technique to improve time predictability of cache memory. However, it can lead to inferior performance because once an instruction or data block is locked, other instructions or data blocks cannot use that cache line dynamically. For this reason, in most cases cache-locking is a trade-off between performance and predictability. Either the cache is not locked and is dynamically utilized by a replacement algorithm or it is locked and becomes static and holds a limited amount of data or instructions and as a result either predictability or performance is compromised. In this chapter we propose a hybrid cache-locking that uses both of these techniques to improve performance and predictability by partitioning the cache so that part of it is lockable and the other part functions normally. This method can be further improved by using benchmarks to determine which cache partition size works best for each benchmark.

3.3 HYBRID CACHE LOCKING

The hybrid-cache-locking concept is a blend between cache-locking and a normal cache. While cache-locking is great for time predictability, it is not usually good for performance. Regular cache with no locking is the opposite, as it has good performance in terms of execution cycles, but is unpredictable. Hybrid cache combines both of these to get the benefits of both techniques. Like cache-partitioning [11], [13],

it allocates a certain percentage of the cache for each technique, but unlike cache-partitioning, there is no actual physical partition. As shown in Figure 3.1 the locked cache can go wherever it needs to in the cache. This makes the hybrid method very exible and allows it to take advantage of associativity to achieve good performance and predictability.

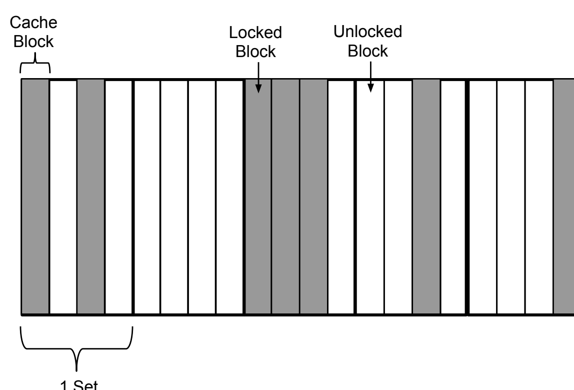

FIGURE 3.1: Hybrid cache-locking.

3.3.1 Locking Methods

There are two locking methods studied in this chapter. The first is a dynamic locking method that determines what is locked at run time. Instructions that are used more times than the threshold number, i.e 16, will be locked into the cache until the cache is full. The second method is a static method where the locked blocks are determined at compile time and it uses only information the compiler has. Instructions that are definitely going to be executed in every iteration of loops are selected to be locked into the cache.

3.3.2 Implementation

To implement hybrid cache-locking in hardware, an additional bit per cache block needs to be used to keep track of the locked state. When the replacement algorithm looks for a cache block to replace in a set, it will skip any blocks that have the lock bit enabled. If it does not find an unlocked block, it will need to fetch the information from the main memory and pass it straight through without storing it in the cache. In addition to the basic cache-locking hardware changes, hybrid cache-locking needs a counter to keep track of what percentage of the cache blocks have been locked. The number of locked blocks will be controlled by checking the counter and comparing it to the number of allowed locked blocks. This control occurs when a block is being locked and the counter is then updated.

In this work, the hybrid cache-locking is evaluated and compared by varying the percentages of lockable cache. Since each benchmark is different, it was not known what percentage of locked cache would be the most optimal in terms of both performance and time predictability. Each benchmark was tested with ten different percentages of locking to compare performance, time predictability and trends the benchmark. Comparisons were also made between benchmarks.

3.4 EVALUATION METHODOLOGY

Like in Chapter 2, the program SimpleScalar [9] was modified and used with the benchmarks from the Mälardalen WCET research group [11]. Two pieces of data are needed

for each ATF data point; the dynamic execution time and the static execution time. SimpleScalar was modified to provide the dynamic execution time by running simulations using the hybrid cache-locking methods. SimpleScalar's configuration is shown in Table 2.1 of Chapter 2. The memory hierarchy consists of only one cache, a Level 1 instruction cache, with the configurations shown in Table 2.2 of Chapter 2. Note that there is no data cache and branch prediction is perfect so that only the instruction cache being studied influences the ATF.

3.4.1 Locking

Dynamic-locking was implemented in SimpleScalar by using a counter that kept track of the number of accesses for each block and would lock into the cache a block whose accesses exceeded the threshold. Static-locking was done by a small program which would decide what blocks should be locked. This program analyzed the code using the control flow graph to find instructions that existed inside loops but not inside **if** statements. It did this to make sure that the instructions locked would be instructions that were definitely going to be executed every iteration. SimpleScalar used the output from the analysis program to check which blocks should be locked when they are accessed.

3.4.2 Static Prediction

The method of static prediction is shown in Algorithm 1 of Chapter 2. The algorithm is similar to an algorithm for simulation; except that the static prediction only uses

information that can be acquired by the compiler. For example the predictor knows information about the pipeline and function units, but cannot predict what instructions will be in the cache unless they are locked into it. The predictor is given the percent of lockable cache so that it can accurately predict how many cache blocks will be locked.

3.5 RESULTS

The *edn* benchmark in Figure 3.2 represents a common trend where the cycle count increases with the percentage of locked cache. There are other benchmarks, *adpcm* and *cnt*, shown in Figures 3.3 and 3.4 that also exhibit this trend as well as *expint*, *ludcmp* and *matmult* which are not shown. These benchmarks have more sets of repeating instructions than what the cache can hold so the instructions get replaced in the cache before they are used again, resulting in cache misses. These benchmarks typically have a set at the beginning that fills the lockable part of the cache. Then the rest of the sets of instructions result in all misses because the sets are too big for the remaining space that is unused by the locked blocks. Also in the case of *edn* and the other benchmarks, ATF only increases because the miss rate increases which then causes the increase in cycles. In these cases the hybrid cache-locking is still useful if predictability is more important than performance and the performance is allowed to be decreased. The hybrid cache-locking can control the extent of the performance decrease and the ATF increase.

The *bsort* benchmark sorts an array of 100 elements and consists of a large loop that does not fit completely into the cache. In Figure 3.5 the initial effect of the locking

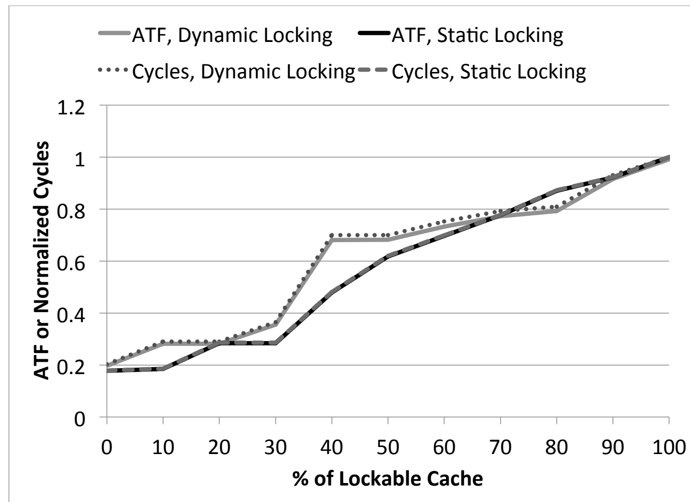

FIGURE 3.2: ATF as percent of lockable cache for *edn* benchmark. Cycles are normalized to 100%

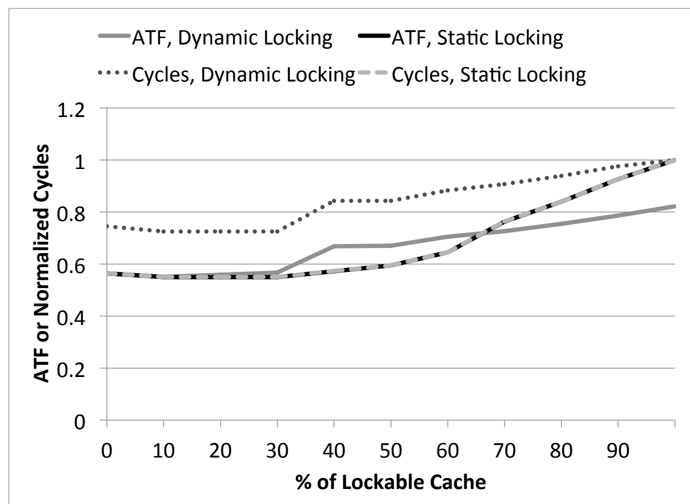

FIGURE 3.3: ATF as percent of lockable cache for the *adpcm* benchmark. Cycles are normalized to 100%

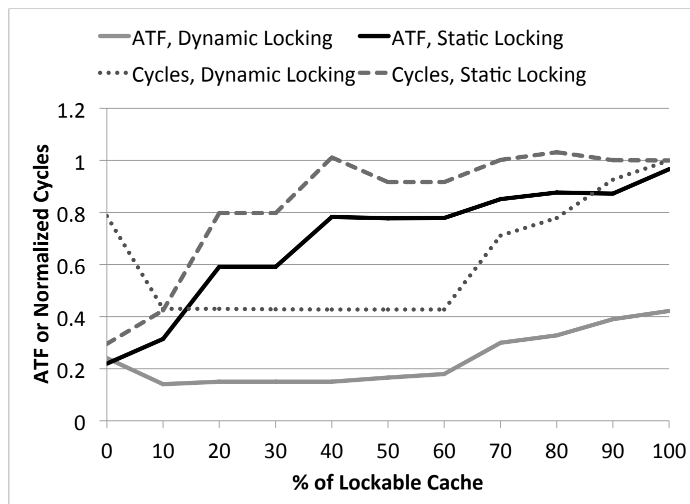

FIGURE 3.4: ATF as percent of lockable cache for the *cnt* benchmark. Cycles are normalized to 100%

method is observed as a decrease in the number of cycles. The locking method guarantees that at least some blocks will be hits every loop iteration. The key observation with this benchmark is that the ATF of the dynamic method continues to increase while the performance stays the same as more cache is made lockable. The static-locking method has a sharp increase in cycles, which by itself causes the ATF to increase and is not considered useful, but after 70% lockable cache, the cycles decrease while the ATF continues to stay high or increase even more.

The *fdct* benchmark in Figure 3.6 is a good example of how cache-locking can benefit performance in certain cases. In this case it is because the benchmark contains a large loop that does not fit into the cache and every instruction is replaced before it can be used again. Cache-locking locks a chunk of that into the cache so it can be

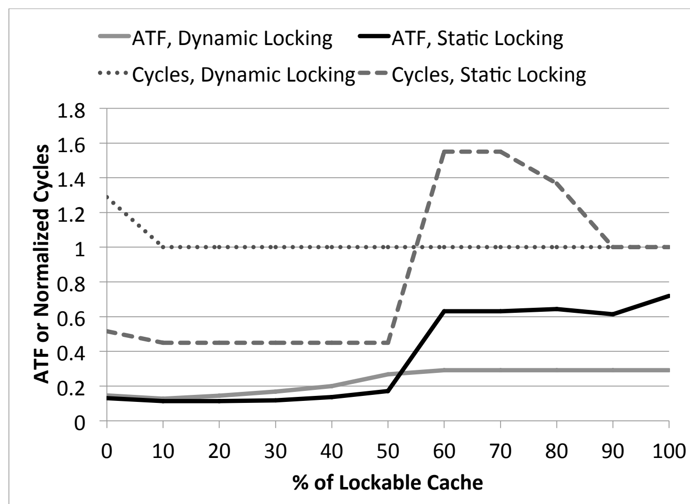

FIGURE 3.5: ATF as percent of lockable cache for the *bsort* benchmark. Cycles are normalized to 100%

reused which causes the performance increase. The static ATF does decrease and fluctuate but only by 0.6% which is small compared to 5.5% reduction of the number of execution cycles.

In Figure 3.7 the *fir* benchmark consists of a large nested loop that has a working set that does fit into the cache. With only partial static-locking, the working set is not able to fit into the cache as well. Starting at 80% to 100% lockable cache, most of the working set is locked into the cache, causing the performance to increase and the ATF to spike. The dynamic-locking method is more flexible, even with only a small percentage of lockable cache. It locks blocks that normally would have been replaced by conditional instructions, causing a performance increase.

The *jfdcint* benchmark in Figure 3.8 is another benchmark that has increasing ATF

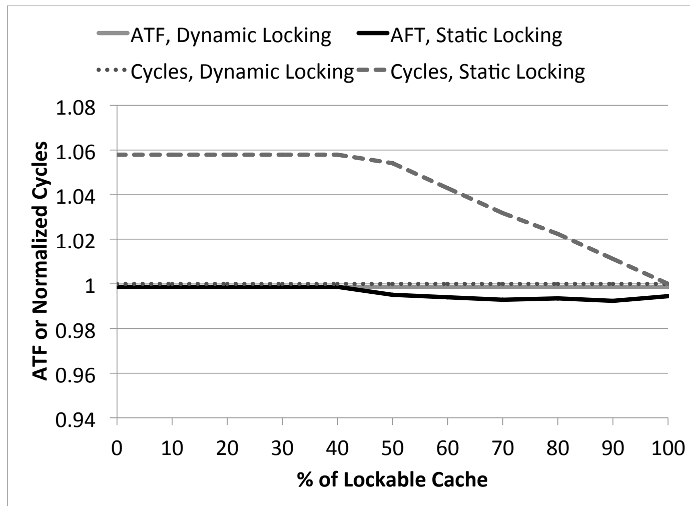

FIGURE 3.6: ATF as percent of lockable cache for the *fdct* benchmark. Cycles are normalized to 100%

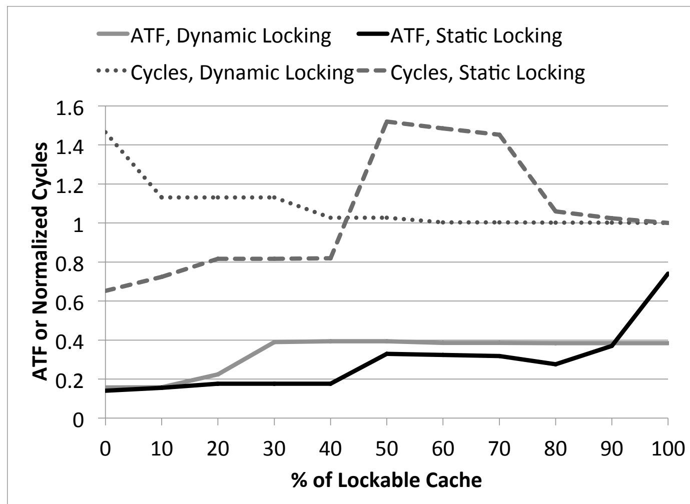

FIGURE 3.7: ATF as percent of lockable cache for the *fir* benchmark. Cycles are normalized to 100%

with cache-locking and does not increase the cycle count, but instead decreases it. The initial effect of cache-locking is noticed, starting at 10% lockable cache, because the lockable cache uses up enough space that a small part of the working set will not be in the cache like it was with 0% lockable cache. *Jfdcint* only has 22 instructions that exceed the threshold for dynamic-locking and, because they were the most used instructions, they would have been in the cache even if they were not locked. The execution time remained the same for both dynamic and static hybrid locking because the cache was large enough to hold the 22 blocks that are locked into the cache when the amount of lockable blocks increased. The fact that the blocks were in the cache even if they were not locked allows for the ATF to increase without any performance cost compared to the other locking percentages.

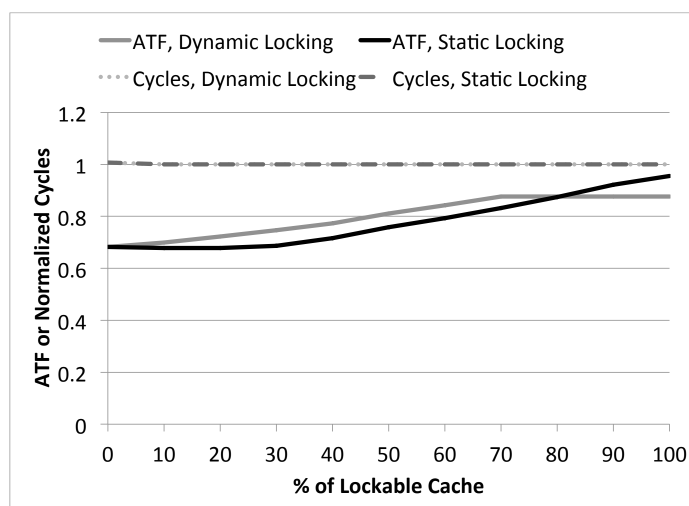

FIGURE 3.8: ATF as percent of lockable cache for the *jfdcint* benchmark. Cycles are normalized to 100%

3.6 CONCLUSION

When comparing hybrid-locking to 100% locking and 0% locking there are trends that are apparent for two groups of benchmarks. First in *bsort*, *fdct*, *fir* and *jfdcint*, 100% locking always has better or equal performance to the best hybrid-locking percentage. In all cases except one static method in *fdct*, 100% locking also has a higher ATF. In contrast, hybrid-locking has better performance when compared to 0% locking for these benchmarks. The second trend seen in *edn* and the other 5 benchmarks not shown, is that the performance is worse at 100% and ATF is at its highest. In this trend, 0% locking is the exact opposite of 100% locking, showing best performance, but worst ATF.

The static-locking and dynamic-locking methods performed differently, especially below 100% locking. Static-locking would usually have a higher ATF, but that would also be accompanied by worse performance. Dynamic-locking was more useful with hybrid locking as the ATF would increase and often it would start increasing before the performance would degrade. Although the ATF does not increase as much as it does in the static method, the fact that when increasing the lockable cache, the ATF increases before the performance degrades, makes that an optimum point for hybrid-locking.

There are two benefits to hybrid cache-locking. The first is the ability to control the ATF and performance by selecting how many blocks can be locked into the cache. The other benefit is being able to observe trends as more blocks are allowed to be locked into the cache. Trends that show increasing performance with the increase in ATF are

not the goal of this paper but the performance and ATF can still be controlled with hybrid cache-locking. The other trends that show where the performance decreases and the ATF stays the same or increases at different percentages of lockable cache are the main reason for using hybrid cache-locking. Hybrid cache-locking gives interesting insight into the performance and time predictability of cache-locking.

Chapter 4 MULTI-CORE CACHE-LOCKING

4.1 CHAPTER OVERVIEW

This chapter describes the performance results for two different locking methods in a variety of multi-core configurations. The static-locking method represents a classic locking method that is decided before run time, usually at compile time. The dynamic method is a completely hardware-based method and, although it has hardware overhead, it is flexible and works with a large variety of programs. Both locking methods focus on only user-space instructions and data and ignore kernel or library instructions. Previously these two locking methods were used to explore single-core cache L1 instruction cache-locking and its effects on performance and time predictability 2, 3. A method of finding the architectural time predictability factor(ATF) was used to put a number to the term 'time predictability'. This number allows the exact cost of time predictability to be calculated in terms of performance.

4.2 MOTIVATION

In real-time systems, cache-locking is used to increase the time predictability of caches. Multi-core caches are different from single-core caches. With multi-core processors, private caches have to have coherency and some caches are shared between several cores. Most of the research has focused on embedded processors that are very simple because these simple systems would be the most predictable. General purpose multi-core processors are being used more and more in areas that have previously only used embedded systems. Some of these areas have a real-time aspect such as a cell phone that needs to be able to process audio and communication in real time. Most of the time a general purpose processor is able to handle these tasks without an issue by using preemption and other techniques, but the cache is still full of the data from the task that was preempted and the requested data from the new task will all be misses, causing stalls while the data is fetched from memory. A more problematic case is if the real-time task has idle time and another process is running, the other process will end up replacing any blocks in the cache that belong to the real-time process, causing misses when the real-time process wakes up. In this case, cache-locking can be very useful. There are two relevant questions that need to be answered about cache-locking before it is feasible to be used on a general purpose multi-core processor. The first is ”How much performance has to be sacrificed?” and the second is ”How much time predictability is gained?”. This chapter focuses on the the first question.

4.2.1 Relevant work

The following papers are relevant to the question of multi-core-cache time predictability. The first paper has a good related works section that discusses other single-core-related methods of cache-locking. It also focuses on using a way-locking method and looks at its performance and time predictability in terms of task delay. [14] Way-locking is very similar to the locking methods used in this chapter. The second paper discusses the benefits of partition-based locking and compares core- and task-based partitioning. [15] The third paper proposes a miss table at the L2 cache level to dynamically and intelligently decide what cache blocks should and should not be locked, based on the number of misses. [16] L2 cache-locking is also explored in this paper. The fourth paper proposes a solution to keep task migration from making cache-locking useless by transferring the contents of the cache and re-locking locked blocks. [17] In our work we do not consider this issue because our tasks are locked to specific cores and can not migrate, but this is a relevant issue for cache-locking on general purpose multi-core systems.

4.3 LOCKING METHODS

4.3.1 Dynamic Locking

The dynamic-locking method implemented in this chapter keeps track of every instruction accessed during the program's run time. When the accesses pass a threshold, the

instructions or data are locked into the cache. While this is not feasible for hardware implementation because it is not possible to keep track of every instruction or every block of data, this implementation shows the best possible choices that the method can make. The actual hardware implementation would only be able to keep track of a subset of instructions, not unlike what is described as a L2 miss table in [16].

4.3.2 Static Locking

Static locking is a locking method that is implemented at compile time by analyzing the control flow graph. Using the control flow graph, instructions and data are selected to be locked in the cache based on two conditions: the instructions or data have to be inside of a loop and they can not be within a conditional statement. In other words, they have to be executed every iteration of the loop. This is best done at compile time by the compiler. The goal of this method is to lock instructions that account for the majority of the dynamic instructions. Once these instructions are identified, they can be locked into the cache in a variety of ways. The best implementation is to have an extra bit that flags whether an instruction gets locked into the cache when it is inserted. The benefit of waiting until the cache block is inserted to lock cache blocks into the cache, instead of preloading them, like some locking methods do [16], is that the actual execution path does not have to be known. Only the blocks that are used for that path will be locked into the cache and, although this causes the first access to always be a miss, it is a predictable miss.

4.3.3 Effects on Time Predictability

Effects of Level 1(L1) cache-locking depend on how many instructions the program has or how much data it uses. If all of the instructions can fit into the cache, then there will be no cache misses after they are locked in. If not all of the data or instructions can fit in the cache, then everything that can not fit will always be misses. In these latter cases the larger, but slower, L2 cache can be useful.

The effect of Level 2 (L2) cache-locking is different than L1 cache-locking. This is largely due to the the fact that the access pattern for the L2 cache is different because the L1 cache absorbs most of the repeated requests for a cache line. Another factor in the effectiveness and performance of L2 cache-locking is the large size of the L2 cache, which allows more blocks to be locked into the cache. For small benchmarks, this means that the whole benchmark can be locked into the L2 cache. Having a whole program locked into the L2 cache means that the worst-case fetch is not from the main memory, but from the L2 cache. Even when the program does not fit completely into the cache, the dynamic-locking method is reasonably predictable for benchmarks with limited paths, allowing for a small and tight worst-case-execution time.

Having a shared cache between cores can also be beneficial, not only because of coherency where they can be evicted from the private caches, but also because, in many cases the cores will share data and instructions and with a shared cache they only occupy one location, saving hardware space and power. [15]

4.4 IMPLEMENTATION

MARSS x86 is a complete-cycle, accurate, full-system simulator for the x86-64 architecture [18]. MARSS is based on PTLsim and uses the QEMU virtual machine for the full-system environment. Table 4.1 shows the configurations of the cache used for L1 cache locking. Tables 4.2 and 4.3 show the cache configurations used for L2 cache-locking. L2 cache-locking used an unlocked L1 cache of the specified size.

This work implements dynamic-locking by adding the address of each access to a list and, when a cache block is inserted into the cache, the list is checked and accumulates each occurrence of the address. If the address has more than a threshold of accesses, it is locked into the cache.

In this chapter the static-locking method is implemented by creating a dump of the executable file into assembly and analyzing the control flow. The list of blocks that are allowed to be locked is then passed to the simulator to check when it inserts blocks into the cache.

Table 4.1 shows the cache configurations that were tested. To simplify labeling, locking methods are shown by using the first letter and the cache levels are signified as L1 and L2 for the first and second level, respectively. Also dynamic-locking is the symbol D and static-locking is the symbol S. For the dynamic-locking in the L2 Cache there were 3 locking configurations. Locking only instructions is signified with (I), locking only data is signified with a (D) and locking both is signified with an (ID). Every simulation was done using a two core processor.

TABLE 4.1: Simulator Configuration L1 Locking

Locking Method	N	D	S	N	D	S	N	D	S
Private L1 Inst& Data Size	16K			32K			64K		
Shared L2 Size	0M								
Associativity	8								
Line Size	64								
Pending queue size	256								
Coherence	MESI								

TABLE 4.2: Simulator Configuration L2 Locking

Locking Method	N	D	I	ID	S	N	D	I	ID	S
Private L1 Inst& Data Size	16K					32K				
Shared L2 Size	1M									
Associativity	8									
Line Size	64									
Pending queue size	256									
Coherence	MESI									

TABLE 4.3: Simulator Configuration L2 Locking Cont.

Locking Method	ID	N	D	I	S	ID	N	D	I	S
Private L1 Inst& Data Size	16K					32K				
Shared L2 Size	2M									
Associativity	8									
Line Size	64									
Pending queue size	256									
Coherence	MESI									

4.4.1 Thresholds

In Chapter 2 the dynamic-locking method was found to have the best performance and selection with a hit threshold of 16 for the L1 instruction cache. The L2 cache has a much different access pattern and a hit threshold of 4 was used for the instruction-only

locking, 8 for the data-only locking and 8 for both the data and instruction locking. It is not yet known which of the L2 cache thresholds are the best, so these were the ones tested.

4.4.2 Benchmarks

In this paper the parsec 2.1 benchmarks shown in Table 4.4 were used with the *simdev* inputs, which are the smallest set of inputs for each of the benchmarks. [19] The number of static instructions is shown in Table 4.4. Notably *ferret* and *vips* are the largest benchmarks, so the negative effect of cache-locking will be most apparent on them. Although these are not the normal real-time benchmarks they were designed to test different parts of the architecture and processor configuration. Each benchmark is parallelized by using either pipelining or data parallelism and implemented by using p-threads.

TABLE 4.4: Benchmarks

Benchmark	Number of Static instructions
blackscholes	1079
bodytrack	99437
canneal	9199
ferret	552522
fluidanimate	6429
swapoptions	15354
vips	509188

4.5 RESULTS

In Figure 4.1 we see that the normalized value for the statically-locked core zero for cache size 16k is much larger than any of the others. For the *blackscholes* peak the miss rate goes from 0.15% to 6.25%. The *ferret* benchmark stands out because both 16 and 32K static-locking methods cause the cache to fill up quickly and therefore have larger performance hits. Figure 4.2 shows the normalized cycle values for static-locking instructions in the L1 cache. Here you can see the effect of the cache miss rate is not as pronounced as it appeared when comparing the normalized cache miss rates. One benefit of this processor is that other parts of the processor, like the superscalar out-of-order architecture, cover up the cache misses. For example *blackscholes* had a 400 times worse miss ratio, but only 1.1 times worse cycle time.

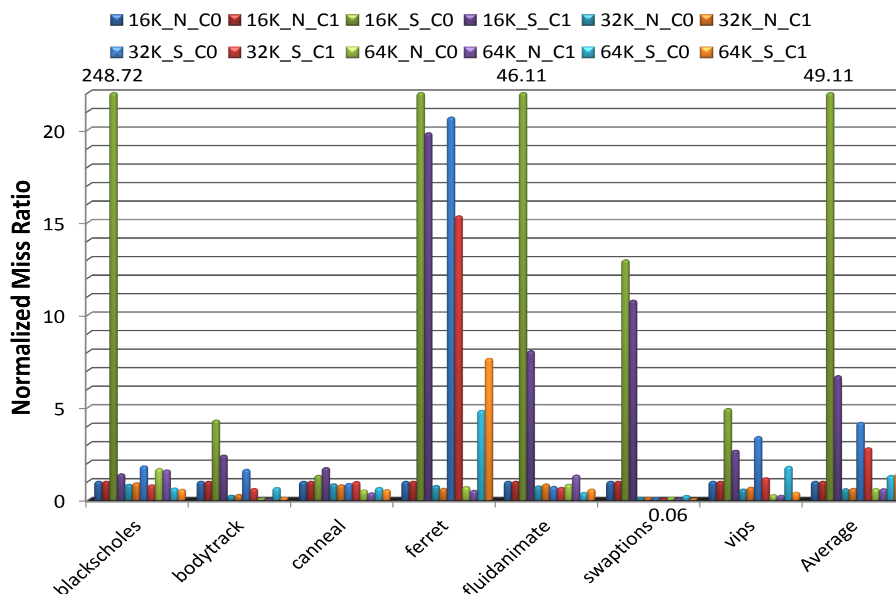

FIGURE 4.1: L1 Static-instruction locking, for size 16k, 32K, and 64K cache, showing the normalized cache miss ratio. Normalized to unlocked 16K cache miss ratio.

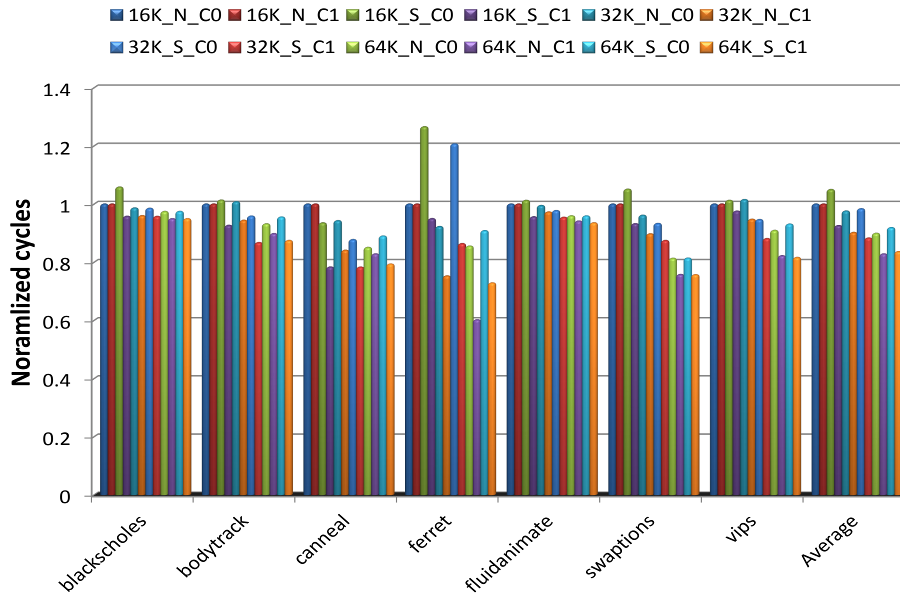

FIGURE 4.2: L1 Static-instruction locking showing the number of execution cycles for 16K, 23K and 64K caches, normalized to unlocked 16K cache.

Figure 4.5 shows the performance of only locking the data in the L1 cache via the static-locking method. The static-locking method affects the data-cache-miss-rate about the same for each of the benchmarks. Most of the benchmarks have a large amount of data to process and it becomes apparent that, as the static-locking method fills the cache, the miss rate substantially increases. Figure 4.6 shows the normalized cycles for the previous configuration. Here we can see that data locking has a much larger effect on the number of cycles. This is because there are more data accesses for each data block then there are for instructions, so the consequences for a cache full of irreplaceable blocks is much costlier.

In Figure 4.3 the performance of dynamic-instruction locking can be observed and Figure 4.4 shows the resulting cycles for this configuration. In Figure 4.3 we see *ferret*, one of the largest benchmarks does not perform well with cache-locking. Looking at

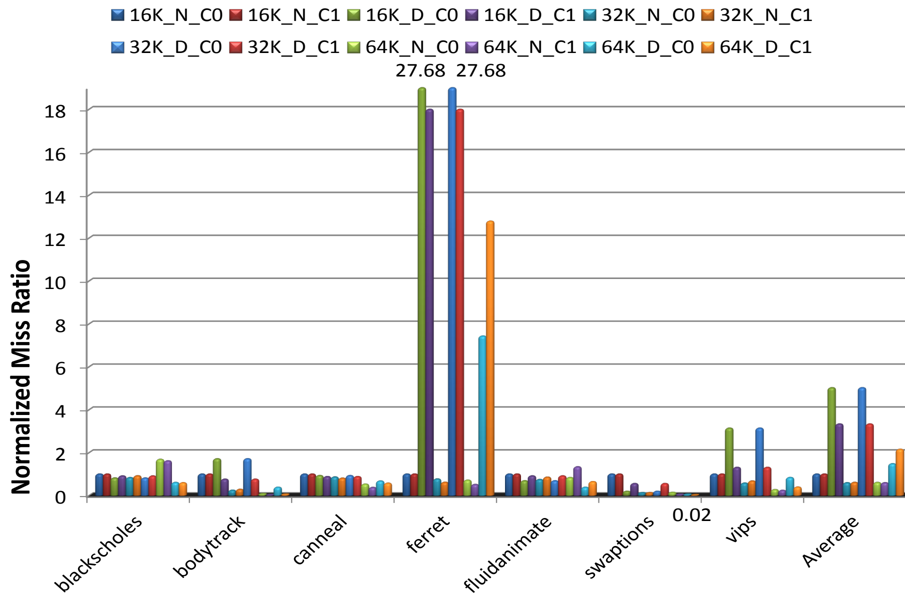

FIGURE 4.3: L1 Dynamic-instruction locking, for size 16K, 32K, and 64K cache, normalized to 16K unlocked cache.

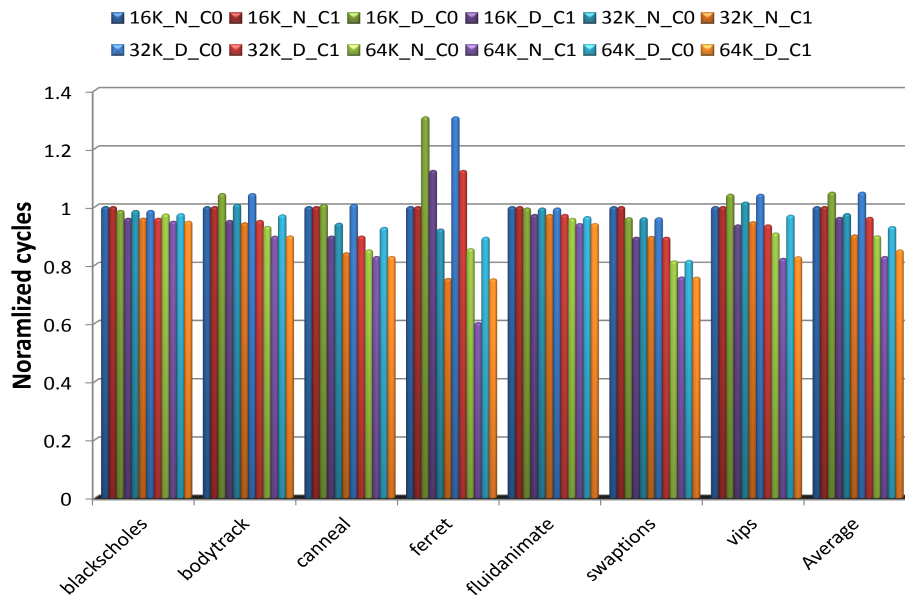

FIGURE 4.4: L1 Dynamic instruction locking, the number of execution cycles for size 16K, 32K and 64K cache, normalized to unlocked 16K cache.

blackscholes we can see that the locking methods actually decrease the miss ratio, but the effect here is minimal as both the unlocked and the locked miss ratio is less than 1%. The cycles in Figure 4.4 show that this has no effect on the performance.

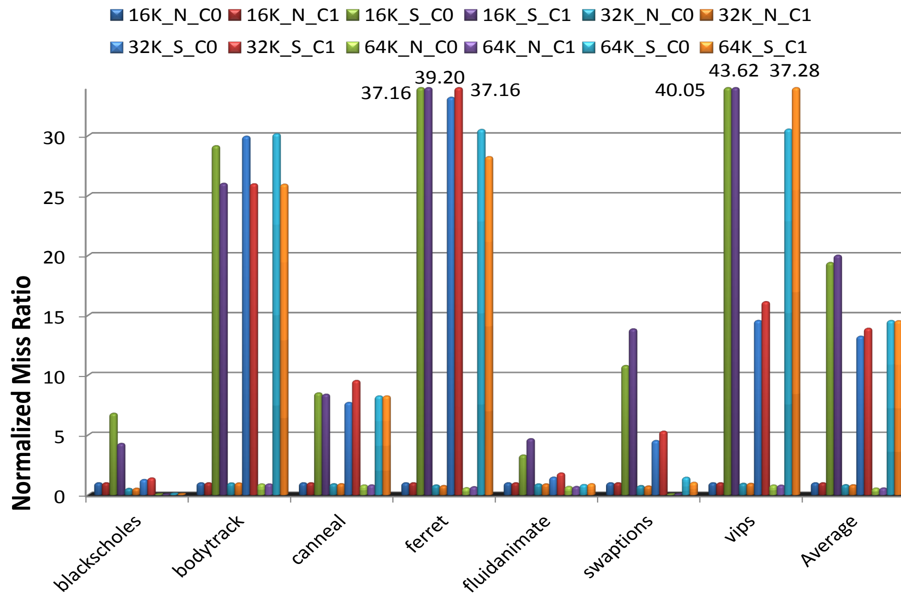

FIGURE 4.5: Miss ratio for L1 static data locking, sizes 16K, 32K, and 64K, normalized to 16K unlocked cache's miss ratio.

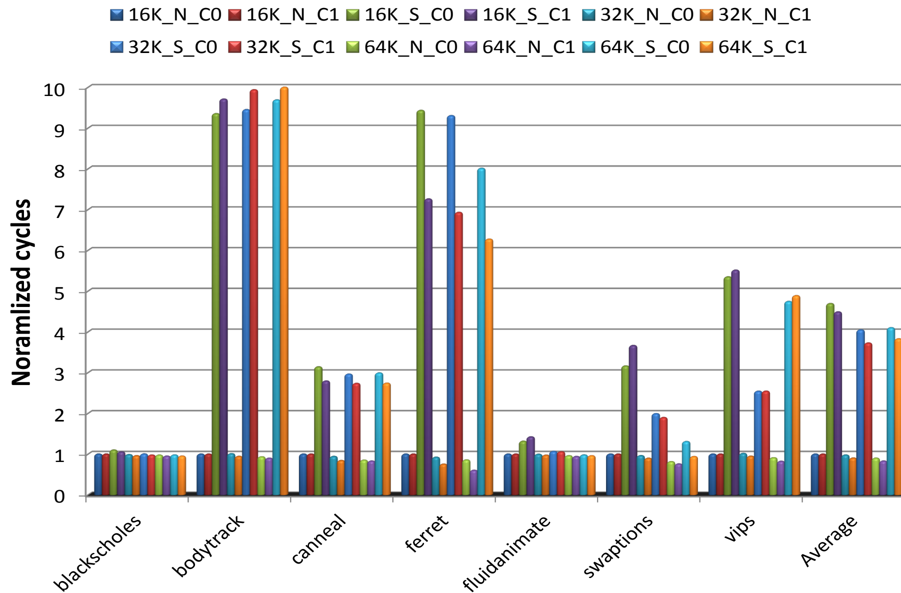

FIGURE 4.6: L1 Static data locking, number of execution cycles for 16K, 23K and 64K caches, normalized to unlocked 16K cache.

A comparison of the static- and dynamic-locking methods for the L1 instruction cache size 32 is shown in Figure 4.7 with the cycle count shown in Figure 4.8. Here we can see that the static-locking method is performing better than the dynamic-locking and in

cases like *blackscholes* and *fluidanimate*, the locking methods improve the miss rate by two to four times. Interestingly, those are not the benchmarks that have improved performance or number of cycles. Once again we see that the miss rate does not solely contribute to the number of cycles as *ferret*'s miss rate was over 30 times worse and its cycle count was only 1.1 to 1.5 times worse. When comparing Figure 4.7 and 4.8 there are two benchmarks that have a worse miss ratio but have better execution cycles, this is most likely due to the out of order nature of the processor and the delay caused by the cache miss allows some other instructions to be processed with less delay. The effect is so noticeable because both benchmarks are large loops that have many iterations and each iteration the effect stacks up.

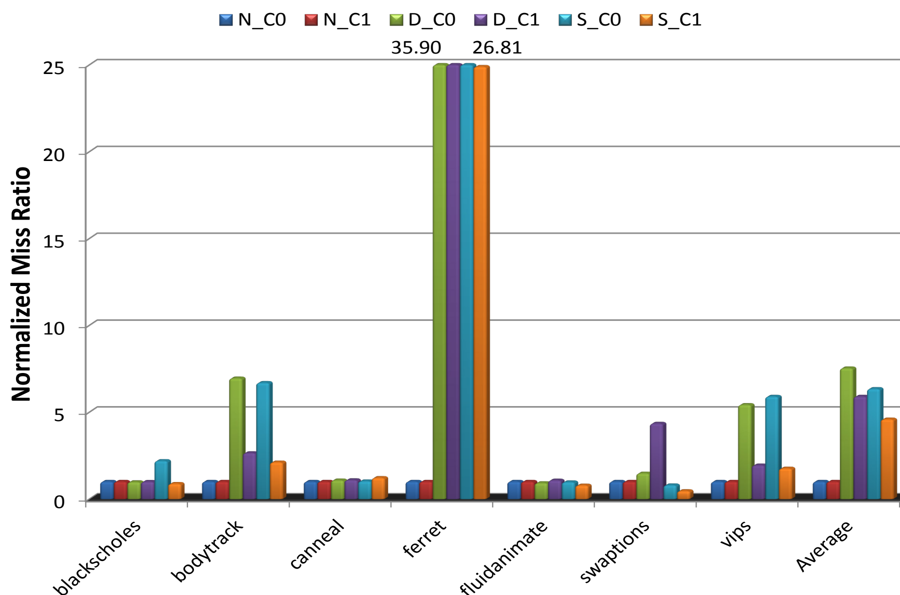

FIGURE 4.7: Miss ratio for L1 instruction locking, size 32K cache, normalized to unlocked cache.

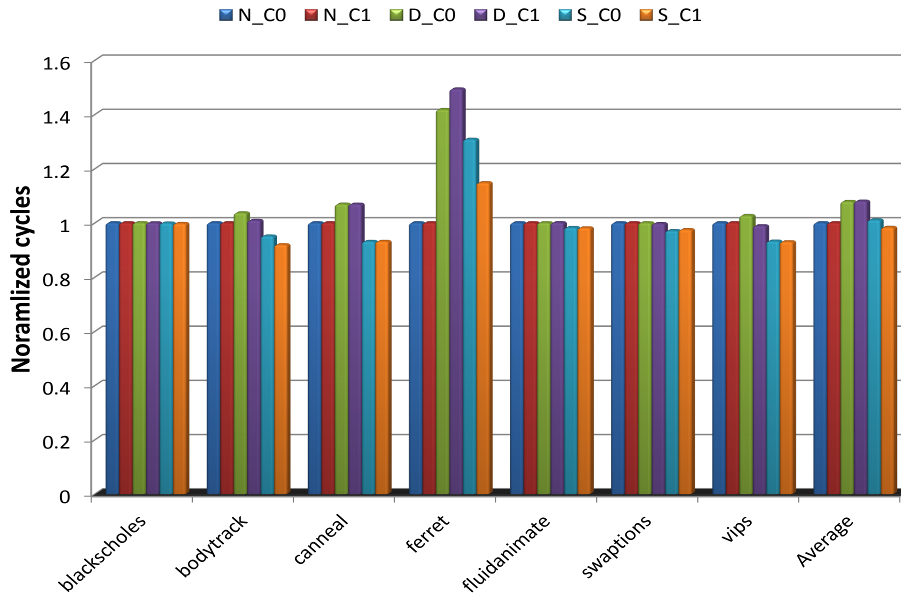

FIGURE 4.8: L1 instruction locking, cycles for size 32K cache, normalized to unlocked cache.

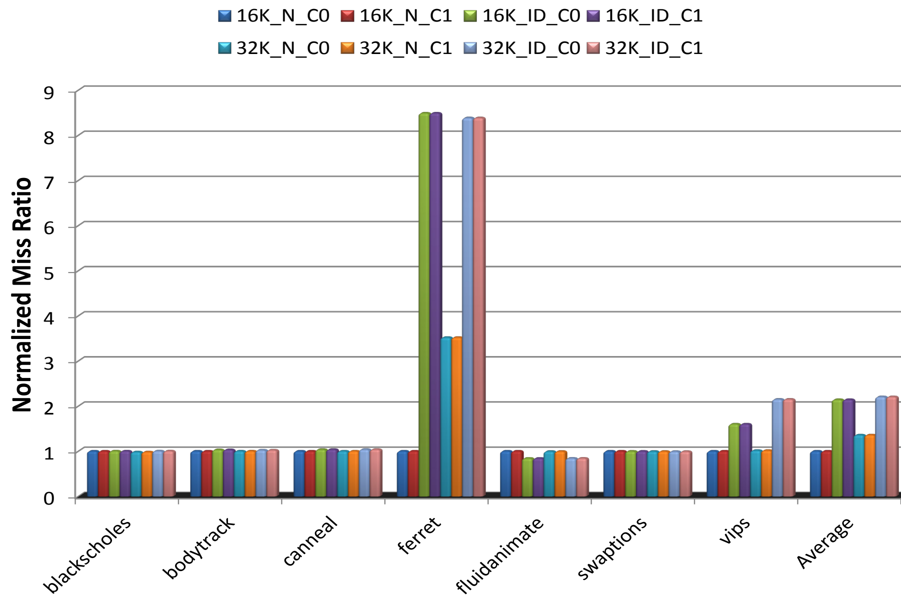

FIGURE 4.9: 16K and 32K L1, 2M L2 dynamic cache-locking, instruction and data, miss ratio normalized to L1 size 16K unlocked cache.

4.5.1 L2 Cache

Figure 4.9 describes the effect of the L1 cache size on a 2M L2 cache and Figure 4.10 shows the effect it has on the execution cycles. In Figure 4.9 it is observed that

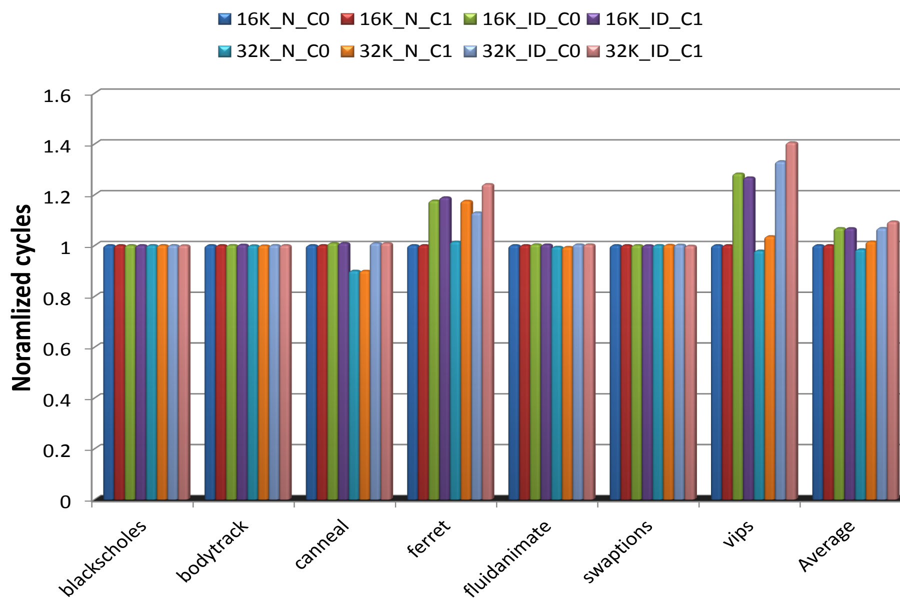

FIGURE 4.10: 16K and 32K L1, 2M L2 dynamic cache-locking, instruction and data, execution cycles, normalized to L1 size 16K unlocked cache.

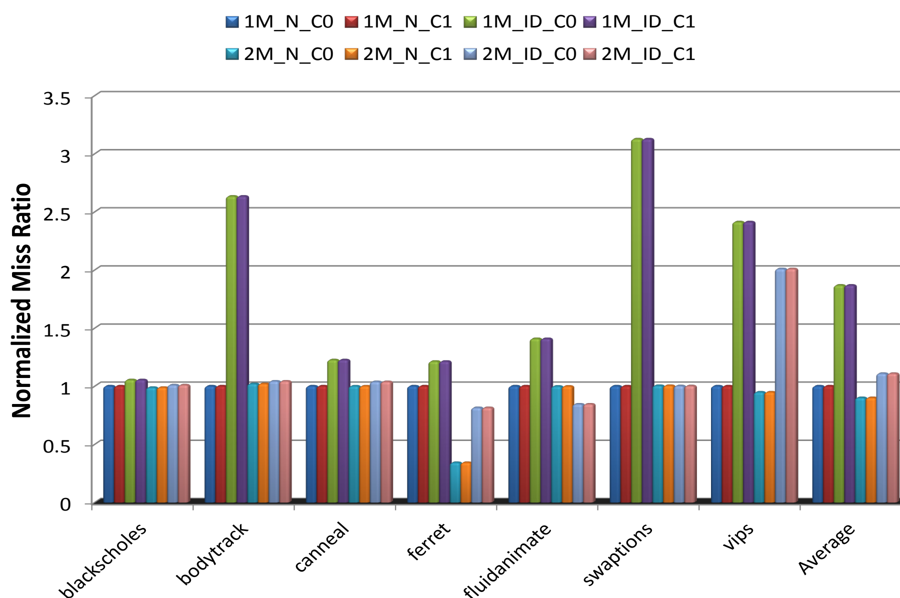

FIGURE 4.11: 32K L1, 1M and 2M L2 dynamic cache locking, instruction and data, miss ratio normalized to L2 size 1M unlocked cache.

changing the size of the L1 cache has minimal effect on the L2 cache. For *vips* we can see that the miss rate actually became worse from 1.5 to 2.1 because the L1 cache was larger and absorbed more of the accesses. Looking in Figure 4.10 the normalized

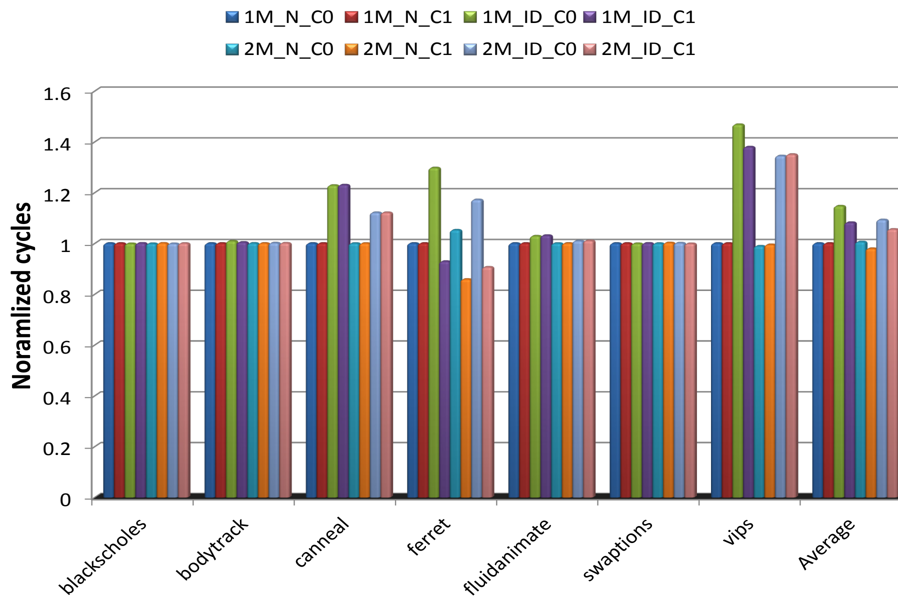

FIGURE 4.12: 32K L1, 1M and 2M L2 dynamic cache locking, instruction and data, execution cycles, normalized to L2 size 1M unlocked cache.

execution cycles increases with the larger L1 cache size for both *ferret* and *vips* due to the increased miss rate of the L2 cache.

Figure 4.11 shows the normalized miss rate of the L2 dynamic cache-locking, locking both data and instructions for different L2 cache sizes with a fixed 32K L1 size. The resulting number of execution cycles is shown in Figure 4.12. In Figure 4.11 we can see that an L2 cache size of 1M is not large enough to hold all of the locked and unlocked instructions and data and performs much like the L1 cache configurations see previously. The 2MB L2 cache size is able to hold all of the blocks selected for locking and have enough free space to allow the unlock cache blocks enough room to perform well in most of the benchmarks. *Vips* is the exception as its data and instructions set is too large for even the 2M shared L2 cache. Figure 4.13 is a comparison between locking only data or only instructions or both in the L2 cache and

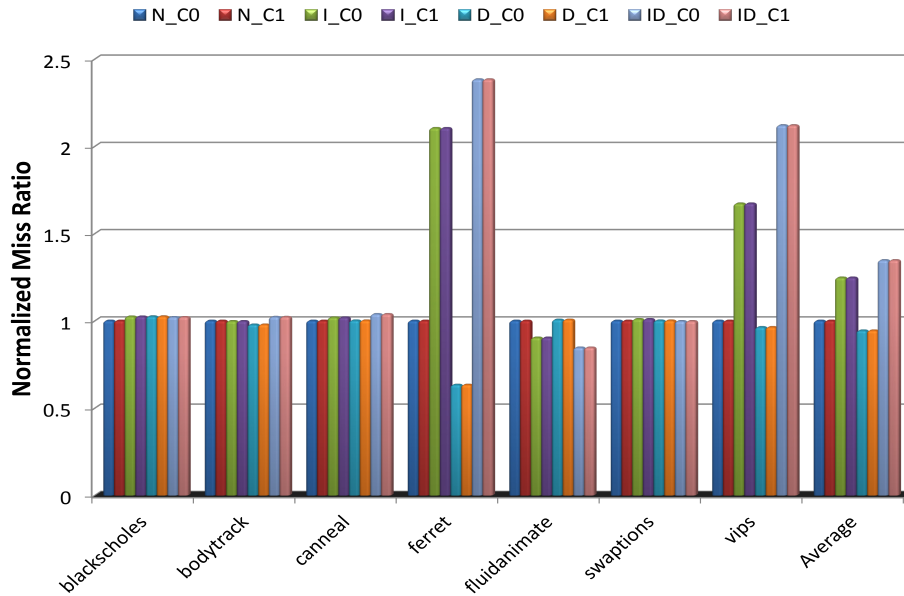

FIGURE 4.13: 32K L1, 2M L2 dynamic cache-locking, comparing instruction, data and both instruction and data locking, miss ratio, normalized to unlocked cache.

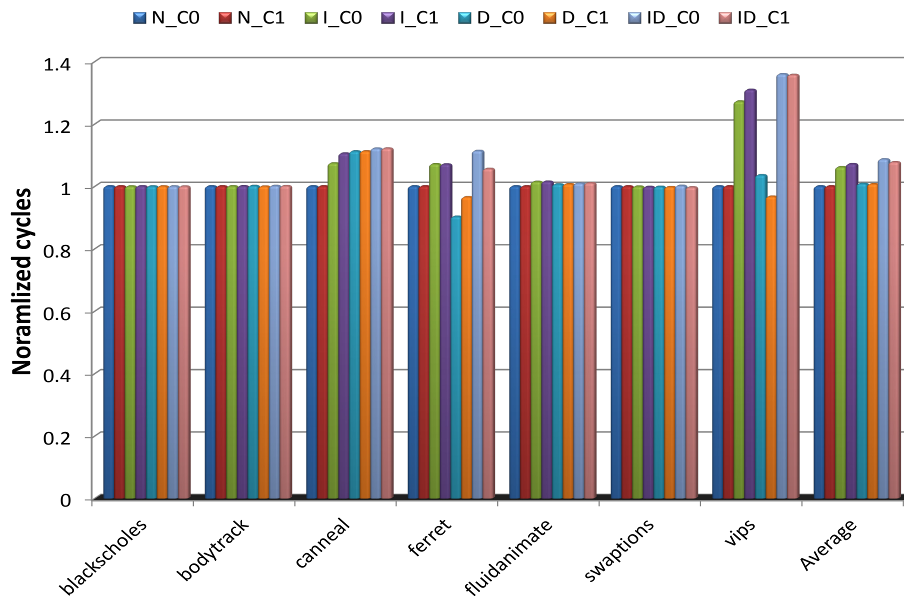

FIGURE 4.14: 32K L1, 2M L2 dynamic cache-locking, comparing instruction, data and both instruction and data locking, execution cycles, normalized to unlocked cache.

Figure 4.14 is the effect on the execution cycles. In Figure 4.13 we see that in most cases, whether the instructions or data or both are locked, the miss ratio stays close to the same for this cache size. Two benchmarks show the difference. *Ferret* and

vips show that there are many more instructions being locked into the L2 cache than data. For *ferret* the instruction locking has doubled the miss rate, a normalized value of 2.1 compared to the unlocked cache, while data-locking has a normalized miss rate of 0.6. *Vip*'s instructions have less of an impact, with a normalize miss rate of 1.67 and a data-locking miss rate of 0.96. Locking the instructions is what causes the largest degradation in performance. This is not the case with all benchmarks. *Ferret* and *vips* are two of the largest benchmarks with the most static instructions. In the cycle graph in Figure 4.14 we notice that *canneal* suddenly shows a degradation in performance to 1.12 that is not noticeable when comparing the cache miss ratios. The Figure 4.17 is

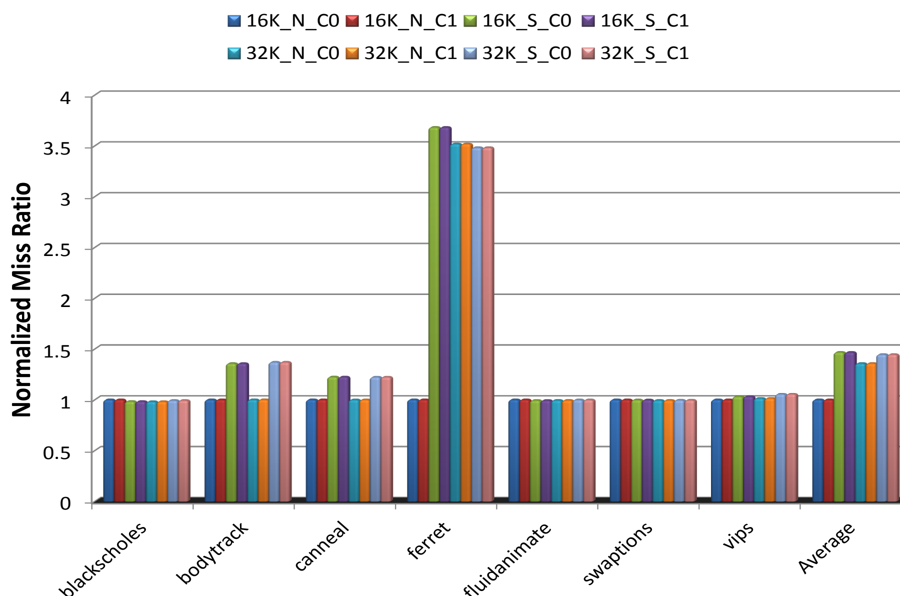

FIGURE 4.15: 16K L1 and 32K L1, 2M L2, static cache-locking, miss ratio normalized to L1 size 16K, L2 size 2M unlocked cache.

the normalized miss rate of the L2 static-cache locking for an L1 cache size of 32K and a varying L2 cache size. As was observed in the dynamic-locking method on the L2 cache, the 1M L2 cache size is not large enough to fit all the locked blocks and have room for the unlocked blocks to achieve good performance, but unlike the dynamic

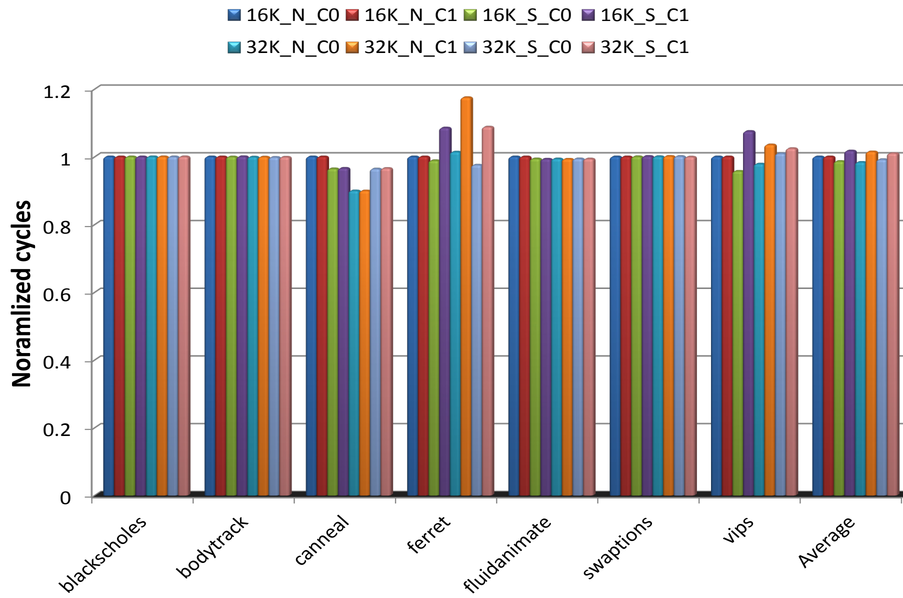

FIGURE 4.16: 16K L1 and 32K L1, 2M L2, static cache-locking, execution cycles, normalized to L1 size 16K, L2 size 2M unlocked cache.

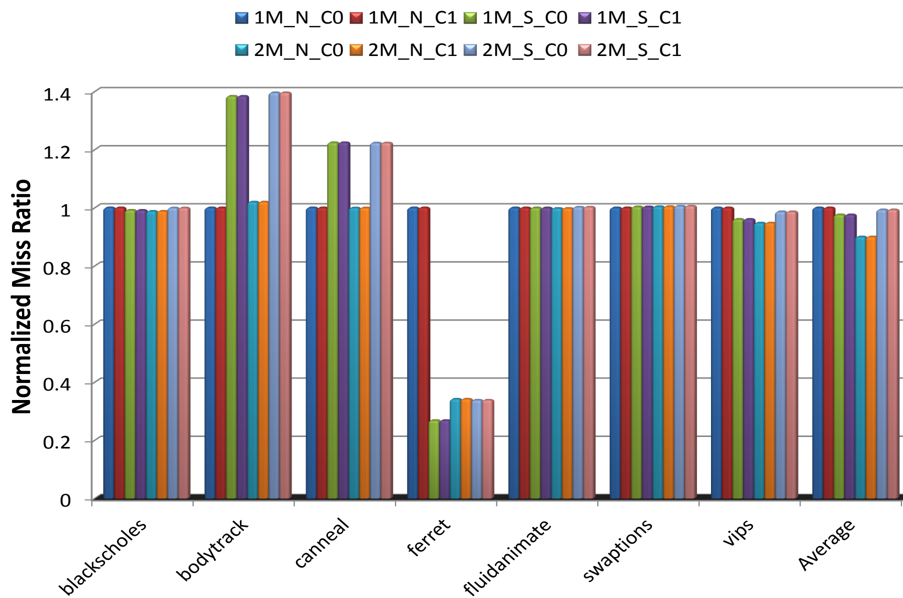

FIGURE 4.17: 32K L1, 1M-2M L2, static cache locking, miss ratio, normalized to L2 size 1M unlocked cache.

locking method, this is only for two of the benchmarks, *bodytrack* and *canneal*. For the benchmarks that show no change, like *fluidanimate* and *swaptions*, this means that everything fit into the 1M L2 cache and the large 2M L2 cache was not fully utilized.

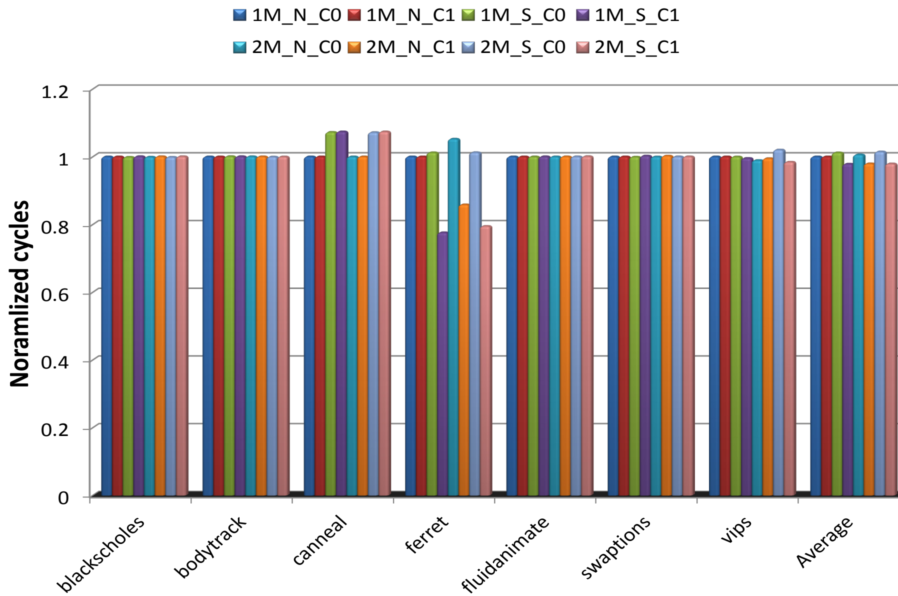

FIGURE 4.18: 32K L1 1M-2M L2 Static cache-locking, execution cycles, normalized to L2 size 1M unlocked cache.

For *ferret* and *vips*, the static-locking method improved the miss rate; in *ferret*'s case from 3.5 to 3.4. This means that key cache blocks were locked into the cache, which prevented them from being evicted, like they would have been without locking. Figure 4.18 show the execution cycles for the various L2 caches sizes with static locking. As with the other caches, the effect of the miss ratio has less impact of the number of execution cycles.

4.6 CONCLUSION

4.6.1 Cache Size

Most cache-performance-related issues are affected by the size of the cache. Cache-locking is no exception and, in fact, it requires a larger cache size than normal, unlocked cache. For general purpose processors that have other applications running at the same time as a real-time application, the cache needs to be large enough to lock blocks for the real-time application and have some free space for the other applications or, other unlocked blocks in general, to mitigate the performance impact. In contrast, in a hard real-time system the full cache would be used to lock as much of the program as possible, since any unlocked block is expected to be a miss anyways due to the WCET estimate. In both of these cases the cache size is directly responsible for the performance; the larger the cache, the better the performance.

4.6.2 L2 Cache

L2 cache is a better solution for performance due to the fact that it does not affect the run time as much as the L1 cache does. The L2 cache had a better hit rate when the L1 cache was smaller because the L1 had more misses than the L2 had in its cache. Locking the L2, instead of the L1, cache helps increase the performance of the average and the worst case greatly. Due to its larger size, L2 is able to fit more instructions and data than the L1 caches, and, depending on the program size, it may have space to

handle unlocked cache blocks, improving the average case performance. Also, since there is an unlocked L1 cache, the access pattern is very different for the L2 cache as the L1 absorbs repeated accesses and the L2 accesses are either L1 cold misses, evicted blocks from the L1 or invalidated blocks. The figures show that a smaller L1 cache is better for the L2's miss rate because the L1 cache does not hold as many blocks or absorb as many cache accesses. This means that the L2's locked blocks are accessed more often. This does dramatically increase the execution cycle count because the smaller L1 cache has more misses.

4.6.3 General Purpose Processor

The MARSS simulator, like most general purpose processors, has many features to cover memory access time. Although there is a limit, as is observed in the data-cache-locking Figure 4.6, the effect of cache misses is reduced, allowing the average case of cache-locking to perform better. This is important on a general purpose system that has other responsibilities besides its real-time applications. Cache-locking on this type of system could bring enough time predictability to allow real-time applications and non-time-sensitive applications to run on the same processor with out a huge performance overhead.

Chapter 5 CONCLUSION

5.1 CONCLUSION

The static- and dynamic- locking methods on many different cache configurations have been intensively explored in this work. In Chapter 2 we explored their predictability and the cost of the predictability. Chapter 3 explored what percent of the cache should be lockable to have the best performance and the best predictability. Chapter 4 investigated cache-locking in general purpose multi-core processors with multiple levels of cache and explored the performance impact.

Chapter 2 introduced the concept of ATF to measure time predictability on a superscalar processor. Two locking methods were tested; static-locking and dynamic-locking. It was observed how the dynamic-locking was flexible and adapted to the different cache configurations and benchmarks. It had an ATF between 0.73 and 0.47 which is better than unlocked cache. Static-locking was more time-predictable with an average ATF of 0.9, but in many cases suffered worse performance. For some benchmarks there were not enough instructions that passed the conditions for the static-locking method to lock in the cache. This caused some degradation of the ATF. The

associativity of the cache improved performance when using cache-locking, but it decreased the ATF. This chapter proposed using the ATF to evaluate the time predictability of these two cache-locking methods on a superscalar processor and succeeded to put a number to how time predictable these architectural features are.

Chapter 3 expanded on Chapter 2 by providing for finer control of the cache-locking methods in the cache by allowing the percent of lockable cache to be specified. Here it was observed that the greatest time predictability is when the cache is 100% cache lockable. There were a few cases where the ATF was higher at lesser levels, but, in most cases, this was due to a large decrease in performance and not an effect of the actual cache-locking method. With hybrid cache-locking, we were able to observe trends in how each of the benchmarks performed with cache-locking. This was a useful feature, but not the purpose of hybrid locking. The purpose was to find a percentage where the ATF and the performance were both as high as possible. For a couple of benchmarks this was the case, but for most of them the ATF only increased with the performance degradation and was most predictable at 100% lockable cache.

Chapter 4 took these two locking methods, static and dynamic, and implemented them on a general-purpose multicore CPU to measure the performance with multi-threaded benchmarks and different cache configurations. We again see that the cache size directly affected the performance of cache-locking; too small of a cache caused the performance to degrade significantly. The L2 cache-locking is a viable possibility for time-predictable cache while maintaining good performance for the average case and

the worst case. Its large cache size allows even the larger benchmarks to benefit from cache-locking without as much of a performance decrease.

5.1.1 Future work

Chapter 4 covers just the performance of cache-locking on general purpose multicore processors. More work still needs to be done to show how much cache-locking on these processors increases the time predictability and at what cost. Currently it is only possible to extrapolate from Chapters 2 and 3 how much predictability is actually gained, but that work does not take into account all of the other unpredictable architectural features the can affect the time predictability.

Bibliography

[1] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347389. URL http://doi.acm.org/10.1145/1347375.1347389.

[2] Isabelle Puaut. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. *Proc. WCET, Vienna, Austria*, 2002.

[3] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for higher program predictability. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 272–282. ACM, 2003.

[4] Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 114–123. IEEE, 2002.

[5] I Puaut and A Arnaud. Dynamic instruction cache locking in hard real-time systems. In *Proc. of the 14th Int. Conference on Real-Time and Network Systems*, 2006.

[6] Abu Asaduzzaman, Niranjan Limbachiya, Imad Mahgoub, and FN Sibai. Evaluation of i-cache locking technique for real-time embedded systems. In *Innovations in Information Technology, 2007. IIT'07. 4th International Conference on*, pages 342–346. IEEE, 2007.

[7] Tiantian Liu, Yingchao Zhao, Minming Li, and Chun Jason Xue. Task assignment with cache partitioning and locking for wcet minimization on mpsoc. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 573–582. IEEE, 2010.

[8] Y. Ding and W. Zhang. Architectural time-predictability factor (atf): A metric to evaluate time predictability of processors. Technical report, Technical Report, Department of Electrical and Computer Engineering, Virginia Commonwealth University, 2012.

[9] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[10] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1):56–67, 2007.

[11] W. Mälardalen. Research group,wcet benchmarks,, 2008.

[12] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28 (2):157–177, 2004.

[13] M. Paul and P. Petrov. Dynamically adaptive i-cache partitioning for energy-efficient embedded multitasking. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(11):2067 –2080, nov. 2011. ISSN 1063-8210. doi: 10.1109/TVLSI.2010.2066995.

[14] Abu Asaduzzaman, Imad Mahgoub, and Fadi N Sibai. Impact of l1 entire locking and l2 way locking on the performance, power consumption, and predictability of multicore real-time systems. In *Computer Systems and Applications, 2009. AICCSA 2009. IEEE/ACS International Conference on*, pages 705–711. IEEE, 2009.

[15] Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, pages 300–303. ACM, 2008.

[16] Abu Asaduzzaman, Fadi N Sibai, and Manira Rani. Improving cache locking performance of modern embedded systems via the addition of a miss table at the l2 cache level. *Journal of Systems Architecture*, 56(4):151–162, 2010.

[17] Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. Predictable task migration for locked caches in multi-core systems. In *ACM SIGPLAN Notices*, volume 46, pages 131–140. ACM, 2011.

[18] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marssx86: A full system simulator for x86 cpus. In *Proceedings of the 2011 Design Automation Conference*, 2011.

[19] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

VITA

MATTHEW RYAN LOACH

Born on September 2, 1988 in Evanston, Illinois.

American citizen

EDUCATION:

Graduated from Glenbrook South High School, Glenview, Illinois in 2007.

Graduated from Southern Illinois University, Carbondale, Illinois in 2011, receiving his Bachelor of Science in Computer Engineering .

RELEVANT EXPERIENCE:

Computer architecture research with Dr. Wei Zhang at Southern Illinois University, fall of 2009.

Internship with the CCDO group at Intel in Hillsboro, Oregon, summer 2011.

AWARDS:

Deans List, Fall 2007, Fall 2009, Spring 2009, Fall 2010

Inducted into Alpha Lambda Delta Freshman Honorary, April 2008

Inducted into Sigma Alpha Lambda National Leadership and Honors Organization, 2010

Achieved Eagle Rank, Boy Scouts, October 2006