2016

Dynamic Level-2 Cache Memory Locking by Utilizing Multiple Miss Tables

Andrew Louis Mocniak
Wright State University

DYNAMIC LEVEL-2 CACHE MEMORY LOCKING BY UTILIZING MULTIPLE MISS TABLES

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

By

ANDREW LOUIS MOCNIAK
B.S., University at Buffalo, 2012

2016
Wright State University

WRIGHT STATE UNIVERSITY

GRADUATE SCHOOL

December 11, 2015

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Andrew Louis Mocniak ENTITLED Dynamic Level-2 Cache Memory Locking by Utilizing Multiple Miss Tables BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Engineering.

_____

Soon M. Chung, Ph.D.
Thesis Director

_____

Michael Raymer, Ph.D.
Graduate Program Director, Computer Science and Engineering

Committee on
Final Examination

_____

Soon M. Chung, Ph.D.

_____

Nikolaos Bourbakis, Ph.D.

_____

Yong Pei, Ph.D

_____

Robert E. W. Fyffe, Ph.D.
Vice President for Research and
Dean of Graduate Studies

ABSTRACT

Mocniak, Andrew Louis. M.S.C.E. Department of Computer Science and Engineering, Wright State University, 2016. Dynamic Level-2 Cache Memory Locking by Utilizing Multiple Miss Tables

Cache memory plays a vital role in a system’s performance by acting as a buffer to quickly supply requested instruction/data blocks from the main memory to the central processing unit (CPU). Cache management techniques may increase or decrease a system’s performance. The results vary from process to process, depending on how well optimized the cache management technique is for a particular process. The inclusion of level-2 (L2) cache locking has been shown in previous studies to be beneficial in increasing a system's performance. This is further improved upon through the inclusion of a miss table (MT), which keeps track of how often each block is missed in the L2 cache.

In this research, we propose the use multiple MTs to keep track of the number of times each block is missed in the L2 cache during the run-time of a process. The information obtained from the multiple MTs are then used to lock the most missed blocks into the L2 cache. This is done by dividing the L2 cache into two partitions: a normal partition and a locked partition. The utilization of this cache memory management technique will aid in the reduction of cache misses.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

ACKNOWLEDGEMENTS

I would like to express my thanks to my thesis advisor, Dr. Soon Chung, for his guidance in the completion of this thesis. My thanks also extend to Drs. Nikolaos Bourbakis and Yong Pei, my thesis committee members. Last on paper but first on my thoughts, I would like to give my most heartfelt thanks to my wife Sara, who diligently provided sustenance on many a long night.

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

Cache memory serves as a buffer between the main memory and the processor. It acts as a temporary buffer for data and instruction blocks that the processor will most likely need in the near future. Cache memory is considerably smaller than the next level of memory, main memory. As a result, it is very limited in the amount of data and instruction blocks it is able to hold. To help resolve this issue, cache memory management techniques attempt to maintain the cache memory by replacing blocks it deems is no longer required in the cache memory with new incoming blocks.

A well optimized cache memory management technique has the potential to improve a system's performance by enhancing the flow of instruction and data blocks from the main memory to the CPU. This improvement is made even more appealing as multi-core processing stresses resource sharing. Thus, maximizing performance while minimizing committed resources is a high priority [2]. Previous research utilizing L2 cache locking has shown this concept of cache memory management technique to improve a system's performance [1, 2].

A cache locking memory management technique is implemented by controlling the evict step of a cache replacement policy. The cache replacement policy seeks to keep only the instruction or data blocks that its driving algorithm declares as being important. A block that is in the cache will be replaced with a new incoming block according to the cache replacement policy being used, regardless of whether the

replaced block will be used in the near future or not. Cache locking protects the blocks from this early eviction.

When cache locking is paired with a miss table (MT), a block is selected to be locked based on how often it is missed. The greater the number of times a block is missed, the greater the performance improvement acquired by locking it [1]. While this is superior to a random selection of a process's blocks to be locked, there exists a possibility that some block locked is not required uniformly throughout the entire life of a process.

1.2 Problem Statement

The inclusion of the intelligent selection of blocks to be ignored by the cache replacement algorithm is a difficult and challenging feat. By introducing cache locking, you are effectively reducing the total size of the cache available to the cache replacement algorithm. This will thus reduce the amount of resources to be shared amongst multiple processes. This runs the possibility of inflicting a negative impact on a system. The risk may be deemed manageable as only a system capable of supplying the optimal amount of cache memory to all processes, or is restricted in the total number of processes able to function at one time, would utilize cache locking.

In this thesis, we present a new cache memory management technique that utilizes cache locking with multiple MTs. We propose the creation of multiple MTs at the start of a process. These MTs will then collect the information about the most missed blocks for their relative portion of a process's run-time. The combination of all these MTs will yield the most missed blocks over the entire life of a process. The L2 cache will then be divided into two partitions: a normal partition and a locked partition. The cache replacement policy for the normal partition is driven by the least-

recently-used (LRU) algorithm, and the locked partition is dedicated to a cache locking scheme.

In order to fully test this technique, the locked partition is tested under three different cache locking schemes. The first cache locking scheme is static and is called *full locking,* which locks the most missed blocks for the entire run-time of a process by using the summation of all the MTs to select which blocks to lock. The second scheme is dynamic and rotates with the most missed blocks for each portion within a process's run-time, such as during a quarter (called *quarter locking*). The third scheme uses a combination of the first and second schemes, and requires the locked partition to be divided into two partitions, one for each locking scheme.

CHAPTER 2

BASIC CONCEPTS ABOUT CACHE AND CACHE LOCKING

2.1 Introduction

Discussed in this chapter is the memory hierarchy and related information pertaining to cache memory and cache locking. The memory hierarchy is essential in that it allows computers to store data on large, slow and non-volatile memory while retaining information for current or near future processing on small, fast and volatile memory. When the memory hierarchy reaches the top (CPU), it once again becomes a tiered design with the inclusion of on-chip memory (cache memory). The management of cache memory is difficult as it is typically considerably smaller than the processes it is running. Because of this, a cache memory management algorithm must be applied which will clear blocks of memory for new oncoming blocks. Cache locking, the securing of blocks of memory in the cache, is a way of protecting blocks that will be needed throughout the life of a process.

2.2 Memory Hierarchy

The memory hierarchy in a computer system is generalized in Fig. 2.2.1. The levels vary drastically in unit cost and access time. Data flow from the bottom level (typically at level-4, hard disk drive) to the top (level-1). At the bottom level, the access speed is very slow while storage space is very large. As the level increases, access speed increases and the amount of data that can be stored decreases. In order to meet the demands of a fast CPU, levels 2 and 3 utilize fast, but volatile, data storage devices. That means, if the power is interrupted to the system, all the data stored at these levels will be lost. Level-2 is usually a part of the CPU, implemented directly

devices. That means, if the power is interrupted to the system, all the data stored at these levels will be lost. Level-2 is usually a part of the CPU, implemented directly onto the same microchip.

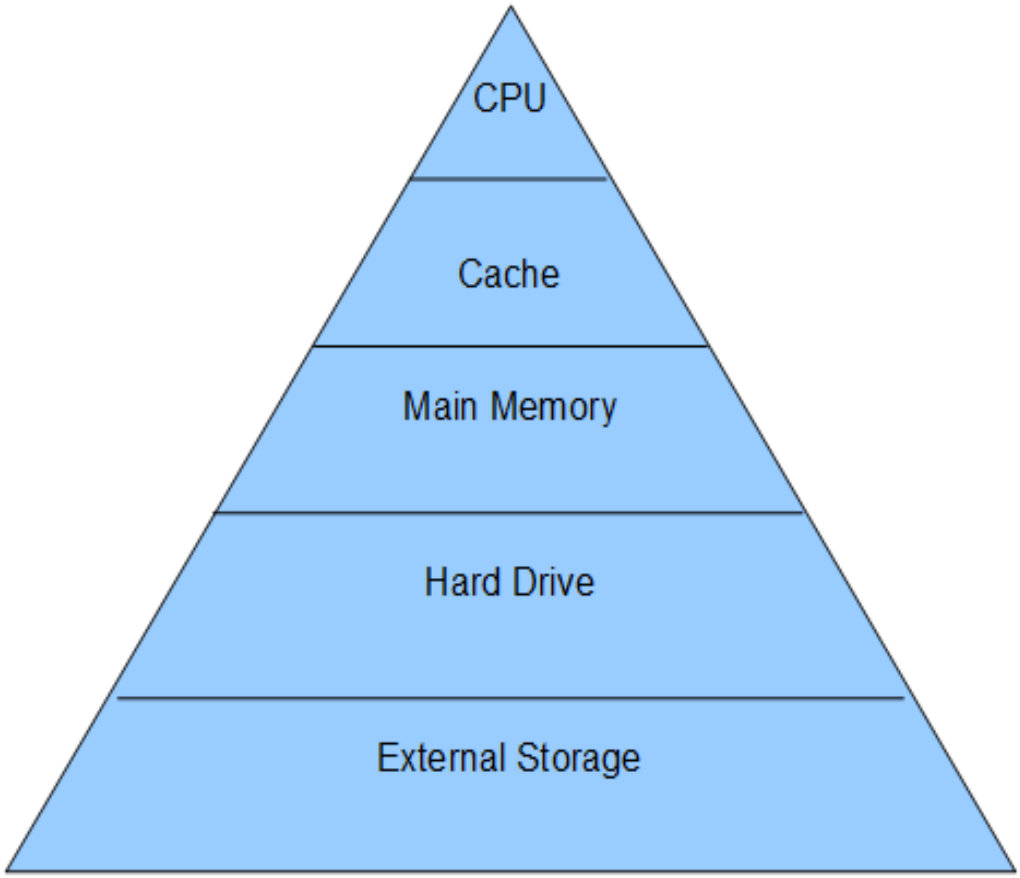

Figure 2.2.1. Memory Hierarchy

Due to the limited storage space available at the levels of cache memory and main memory, their management systems try to keep only the most recent data at those levels. This is done by replacing the data which seem to be no longer needed with newly demanded data.

The replacement of data comes at a heavy cost. A process is delayed as it waits for data to be brought up from a lower and slower level of memory. If some other processes are dependent upon the completion of this process, those processes are also forced to wait.

A replacement algorithm is used to calculate the perceived optimal selection for what data to evict, in order to make room for the new data [3]. Many replacement algorithms are available, but the most commonly used is the least-recently-used (LRU) algorithm. At the cache memory level, the unit of storage is a block, and the LRU algorithm replaces a block that has not been used for the longest time, expecting that block will not be needed in the near future.

The central processing unit (CPU) design affects the structure of its cache memory. CPU designs vary greatly but may be generalized into two groups: single- or multi-core. A single-core CPU has only one processing unit that is able to execute a single thread of a process at a time. A multi-core CPU has two or more processing units to allow for parallel execution of multiple threads, one on each core, at the same time.

Cache memory stores instruction and data blocks fetched from the main memory. Cache memory access time is considerably smaller than that of the main memory. So, the inclusion of cache memory allows for a computer system to perform better by quickly fetching the required word from the cache memory, if it is stored there, instead of fetching it from the main memory.

To improve the performance further, cache memory is divided into multiple levels as shown in Fig. 2.2.2. The number of levels of cache memory is dependent on how many cores a CPU has. A single- or dual-core CPU usually has level-1 (L1) and level-2 (L2) caches. In the case of dual-core, the CPU usually has two L1 caches (one for each core) and one shared L2 cache. When a CPU has more cores, it commonly expands upon this to have a level-3 (L3) cache and possibly a level-4 (L4) cache.

There are two types of blocks in the cache memory: instruction blocks and data blocks. Instruction blocks store instructions to be executed by the CPU, and data

blocks stores the data to be used or modified during the execution of instructions. Since the instructions are fetched to the Instruction unit (*I-unit*) to be decoded, and the data are fetched to the execution unit (*E-unit*) to be used, L1 cache is divided into two sections: *I-cache* and *D-cache*.

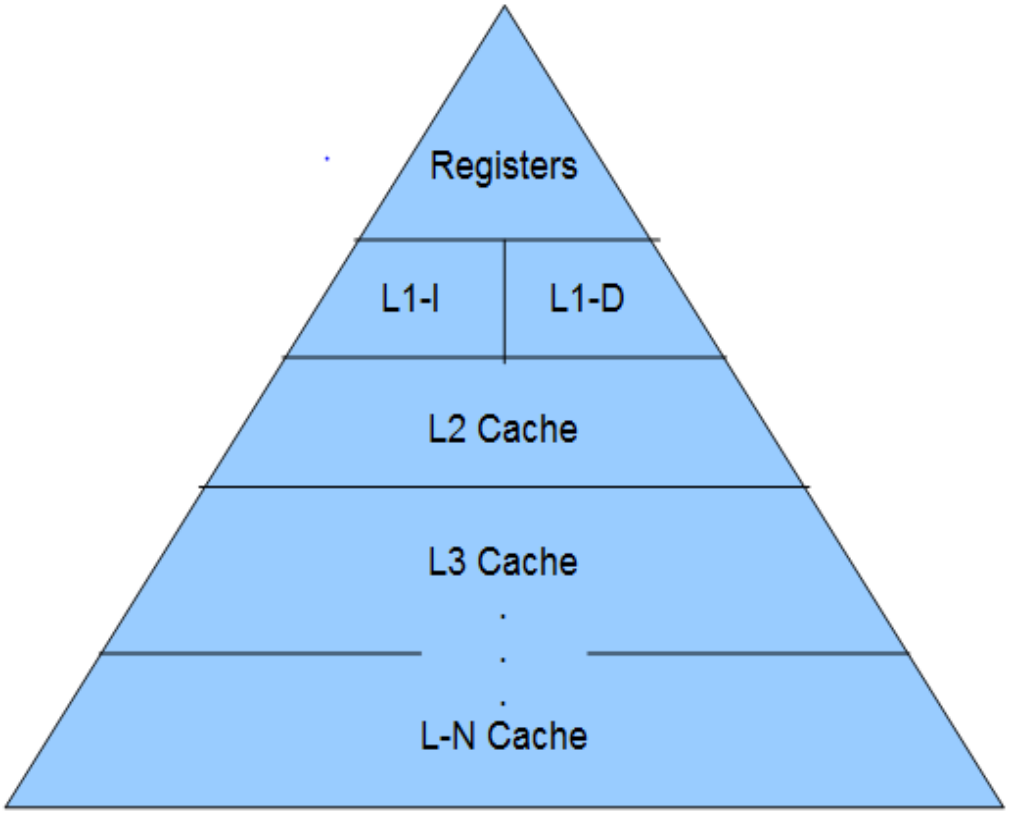

Figure 2.2.2. CPU and Cache Memory Hierarchy

L2 cache is either designated to a single core or shared among multiple cores; L3 cache is almost always shared among two or more cores. Sharing cache memory among multiple cores is a complex procedure as the same block is often required among multiple cores running different threads. The cache management system is responsible for maintaining the consistency of shared data among the cores.

2.3 Cache Memory

Blocks are stored in cache memory with two main components, data and tag. The data is the portion that contains either the instruction or the data to be used by the

processor. The tag portion is extracted from the memory address and is used to identify the block. When the processor sends a block request to the cache memory, it uses this tag to locate the block it is requesting.

For the mapping of blocks between the main memory and cache memory, each level of cache memory often groups its block frames into sets, where each block frame (aka line) can store a single block. This technique is called set associativity, where each main memory block is fetched to a predetermined set, but can be placed in any block frame within the set. As a result, cache memory management system greatly reduces the time and resources required to check if a demanded block already exists in cache.

In a 4-way set associative cache, each set is composed of 4 block frames; and in an 8-way set associative cache, each set is composed of 8 block frames. It has been reported that a larger associative set reduces the cache miss ratio by extending the time a block resides in the cache, as long as it is accessed more frequently than some other blocks stored in the same set.

2.4 Cache Locking

Cache locking is the removal of access of select cache memory blocks from the primary cache replacement algorithm. These "locked" cache memory blocks are instead maintained by a separate, secondary cache replacement algorithm. Despite the term "lock" being used, the data may be replaced as often as the non-locked cache. This term signifies only the separation of select cache memory blocks from the primary cache replacement algorithm.

Different models exist in the selection of blocks to be stored and maintained in the locked cache memory blocks. Assisting tools, such as a MT, may be used to further improve the effectiveness of the locked cache's replacement algorithm. Aiding

tools may collect data to be applied to the current and/or future runs of a specific process. Despite the differences in block selection, all cache locking is inherently alike as the idea across all designs remains the same; the removal of cache blocks from the primary cache replacement algorithm and granted unto a secondary cache replacement algorithm.

CHAPTER 3

EXPERIMENTAL CACHE REPLACEMENT DESIGNS

3.1 Introduction

This chapter will go into detail about each cache locking design that we implement. In total, four designs are used. The first design is our control which utilizes only the LRU cache replacement algorithm. The next three designs incorporate a cache locking scheme.

3.2 Cache design: Control

The first design implemented was the LRU cache replacement algorithm. This design served as the base for all the other designs. The LRU cache replacement algorithm functions by populating the cache with blocks and keeping track of when they were last used in the cache. When the cache is full, this algorithm will find the block that was used the least recently and evict it from the cache. The new block is then fetched and populated into the now vacant space in cache memory.

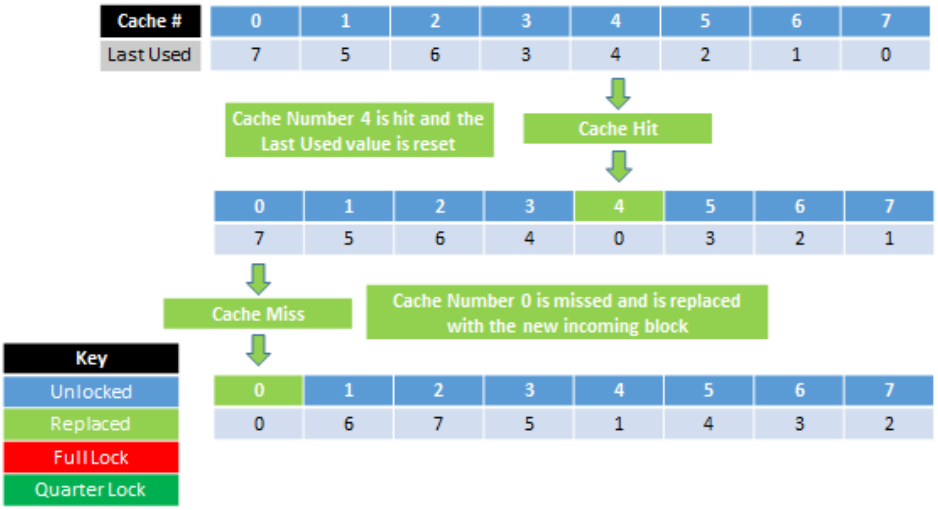

Figure 3.2.1. L2 Cache with LRU Implementation

Figure 3.2.1. shows a L2 cache with an LRU cache replacement algorithm that mirrors the control design. The cache numbers (Cache #) that are dispalyed in blue are not locked and fall under the control of the LRU. Underneath the cache numbers is the Last Used value, which indicates how many block replacements have occurred since that cache block's data was last used. As shown, if a block is hit, it's last used value is reset 0 and all other blocks, which have a last used value less than the block that was hit, are incremented by 1. If a block is missed, the cache number with the highest last used value is selected and replaced with the new requested block.

3.3 Cache design: Test

The second design focused only on the information from the individual MTs (e.g., quarterly MTs). This design dynamically changed the blocks in the locked partition based on new MT information after the respective numbers of block requests were obtained. The third design used the information only from the total MT which is a combination of individual MTs. The fourth design was a mix of both the second and third designs, dedicating half of the locked partition to the individual MTs and the

other half to the total MT.

Before any of the cache locking designs are implemented, a data gathering run is made in which the size of the cache is reduced by the percent of the cache locked. From this modification and data gathering, we can logically select what blocks are most likely to be missed then selectively lock them at their proper stages.

3.3.1 Quarter Locking

In this cache locking design, multiple MTs are used. This is done so that a selection of blocks to lock is different at each stage of a process. A process's stages are defined based on the size of the process (number of block requests to the CPU) divided by the number of MTs (e.g. 100 block requests divided by 4 MTs defines 4 unique stages with 25 block requests in each). At the time of a stage change, all the locked blocks are removed and the new blocks are placed in.

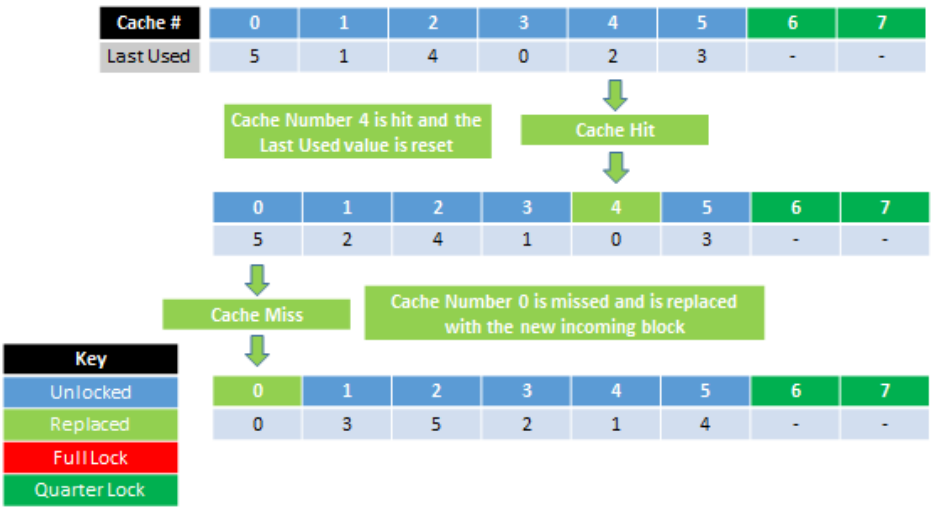

Figure 3.3.1. L2 Cache with Quarter Locking Implementation

Figure 3.3.1 displays an example of this locking design. Cache numbers 0 to 5 are unlocked and under the LRU cache replacement algorithm, with cache numbers 6 and 7 locked. This amounts to a 25% cache locking design. If a cache hit were to

occur on either of the locked blocks, no action is made by the LRU on the unlocked blocks. When a hit or a miss occurs on one of the unlocked blocks, the LRU acts as previously discussed in section 3.2.

3.3.2 Full Locking

Full locking occurs when only one MT is used throughout the life of a process. This means that the data locked is never dynamically changed. Figure 3.3.2 shows an example of this design. The actions that occur on a hit or a miss are the same as explained in the previous sections above.

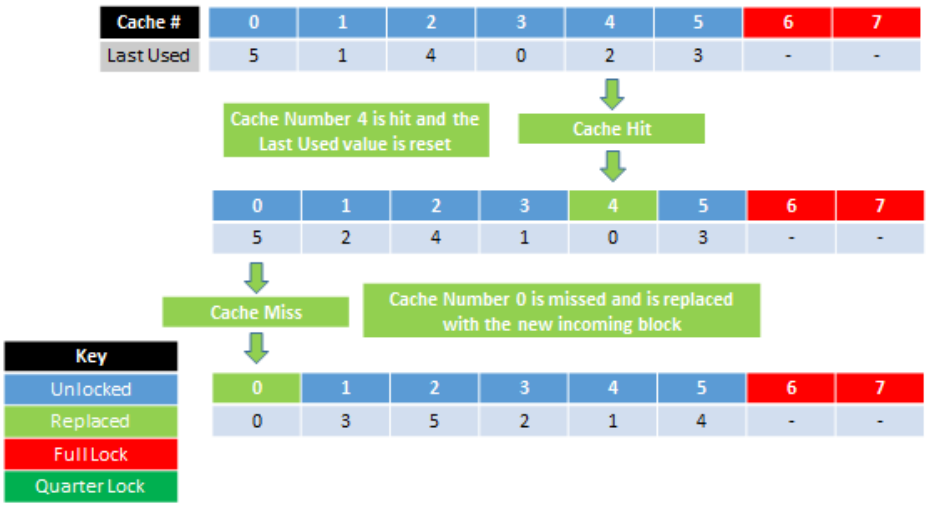

Figure 3.3.2. L2 Cache with Full Locking Implementation

3.3.3 Quarter and Full Locking

Quarter and full locking incorporates both dynamic and static locking designs. The cache dedicated to locking is partitioned, where half is used only for quarter locking and the other half is used for full locking. Each partition operates independently of the others and performs as described in previous sections. Figure 3.3.3 shows an example of this design.

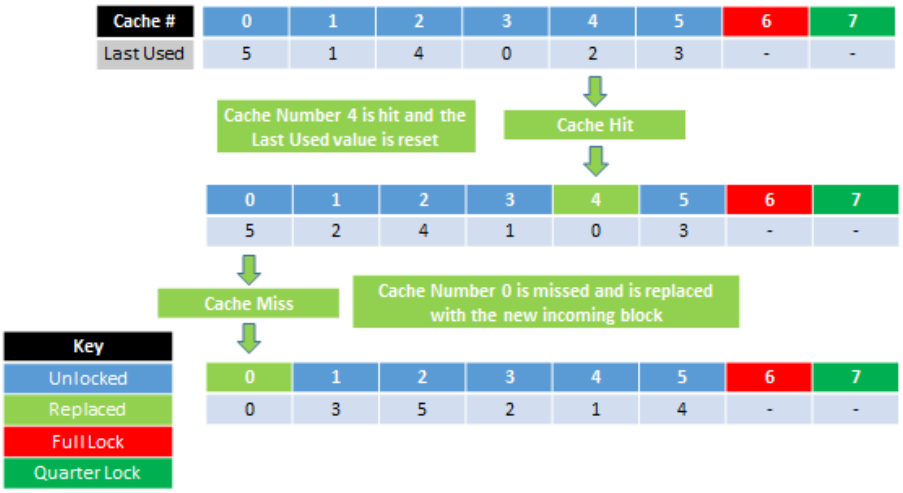

Figure 3.3.3. L2 Cache with Quarter and Full Locking Implementation

CHAPTER 4

SIMULATION

4.1 Introduction

We developed our own unique simulator in C# using Microsoft Visual Studio. By creating this simulator to test our proposed cache locking scheme, we were able to modify and change varying properties in order to better define our design. We compared our simulator against varying workloads and expected results in order to gauge our effectiveness in implementing a L2 cache simulator. After confirming full functionality, we proceeded to implement a modified version utilizing our scheme and compared it to the results of the unmodified version.

4.2 Simulator

A custom-built simulator to test our cache locking technique was designed. This simulator was implemented in C# using Microsoft Visual Studios and was designed to mimic L2 cache with a LRU replacement algorithm.

4.3 L2 Cache Design and Demonstrations

An integer array with the size of the bounded range (cache_range) of the cache was created to act as our L2 cache. The value stored within this array flags the block as being either locked (negative number), inactive (zero) or in use (positive number). The positive value is obtained by an integer counter (count_total) which keeps track of how many requests for blocks have been made to the L2 cache. If a block is requested, it obtains this positive value from count_total if it is either inactive or in use. Another integer counter (count_cache) is used to keep track of how much of the

cache is in use. If a block is already flagged as being in the cache, a cache hit occurs resulting in the count_cache not incrementing and the block acquiring a new count_total number. Once this number has reached the size of our cache (cache_size), we begin to replace blocks by selecting the lowest non-zero and non-negative value stored in the L2 cache array. This victimized block is then set to zero, and the new oncoming block gains the next value from count_total.

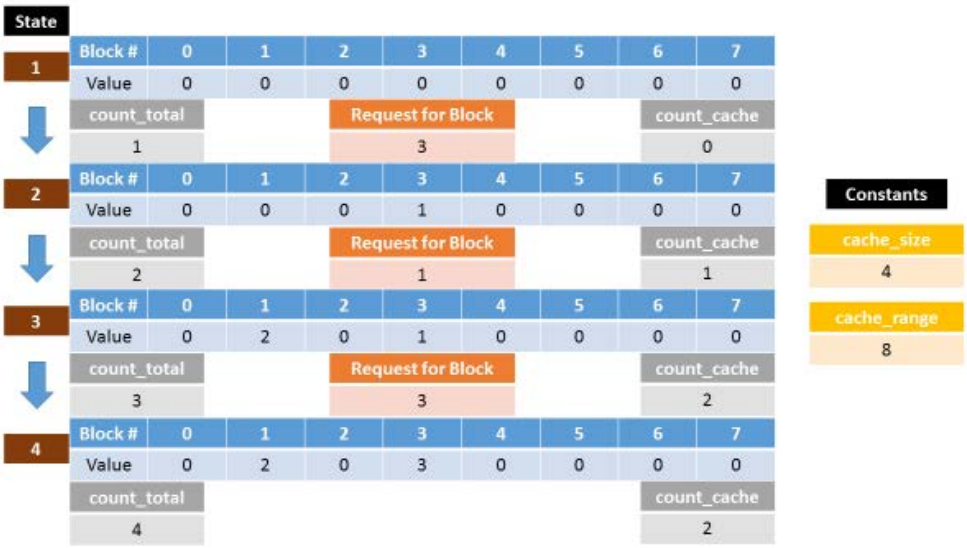

Figure 4.3.1. L2 Cache Array Design in Simulator and Demonstration Part 1

Figure 4.3.1 shows an example of our design in the simulator when it first activates. In the first state, it is empty and receives a request for block 3. The block is found in the array and the value becomes updated with what is stored in count_total, which in turn increments. In the second state, block 1 is requested. The new count_total value is again placed in the array at the proper block with the count_total again incrementing. Note that at this time count_cache has incremented twice, as two blocks are currently considered being held by the cache. In the third state, block 3 is again requested and in the fourth state we can see that it has updated its value, but count_cache did not increment. This is because the block is a positive number, signifying that it already exists in the cache.

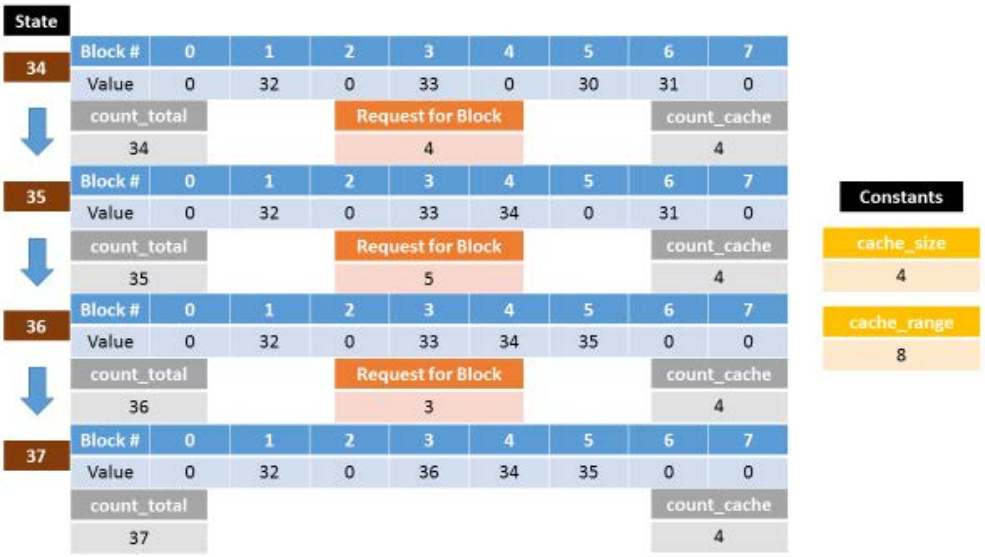

Figure 4.3.2. L2 Cache Array Design in Simulator and Demonstration Part 2

In the next example shown in Figure 4.3.2, we have a cache that has had 33 prior block requests. The cache is considered full, as count_cache is at the value of cache_size. This cache is then pressed with a request for block 4, which is not currently marked as being stored. The LRU selects the lowest value stored, block 5, and zeros out the number. It then goes to the block requested, block 4, and places in the count_total value. In state 35 we can see a request is immediately made for block 5, which has just been cleared. Another cache miss occurs and the process is again repeated, with block 6 being zeroed and block 5 gaining the new count_total value. In state 36 a request is made for block 3 resulting in a cache hit. As can be seen in state 37, the only action that occurs is the update to the value stored for that block.

As stated earlier, in the case of cache locking a negative number is used to mark what is locked. The negative number used depends upon the locking type that is being simulated at the time. When only full locking is being applied (one MT) a -1 is used. When only multiple MTs are used a -2 is used to signify the dynamic locks. The reason for the different negative values is so that we can easily combine the two locking schemes together in our third cache locking scheme.

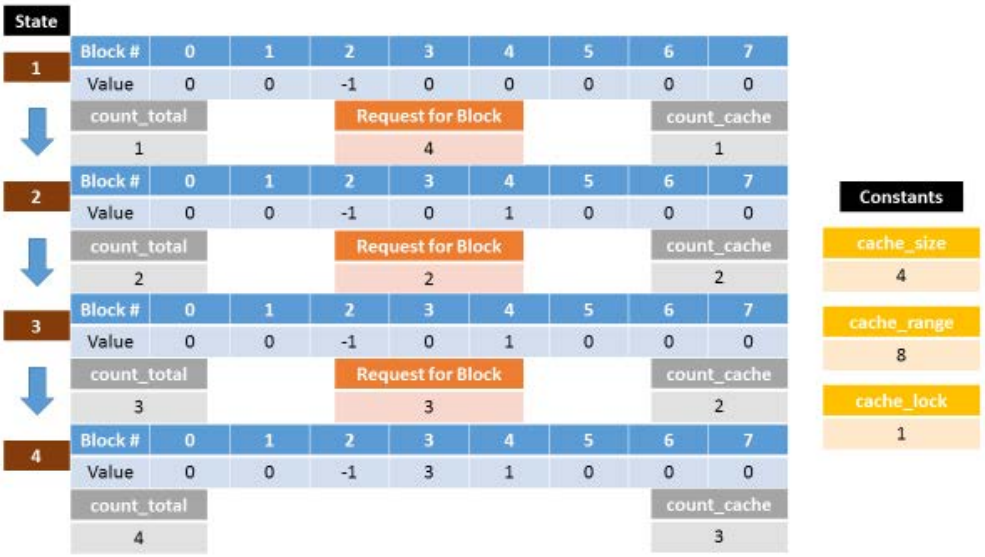

Figure 4.3.3. L2 Cache Array Design in Simulator and Demonstration Part 3

An example of the cache locking design is seen in Figure 4.3.3. To the right of the figure, we can see that this design implements 1 dedicated cache lock. In the first state we can notice that block 2 has been locked as its value is a -1. The cache_size is limited 4, and with the cache_lock already claiming a slot we can see that count_cache is already incremented to 1 to compensate for this addition. This will leave us with only 3 blocks that the LRU has the ability to modify. At state 1, we have a request for block 4 which is inactive. As explained earlier, this block's value assumes the count_total, which in turn increments. Next, we see a request for block 2, which is currently locked. Given this request, no action occurs except that the count_total is incremented by 1. This number is still incremented as it plays a vital role in detecting when to dynamically change locks in the case of multiple MTs.

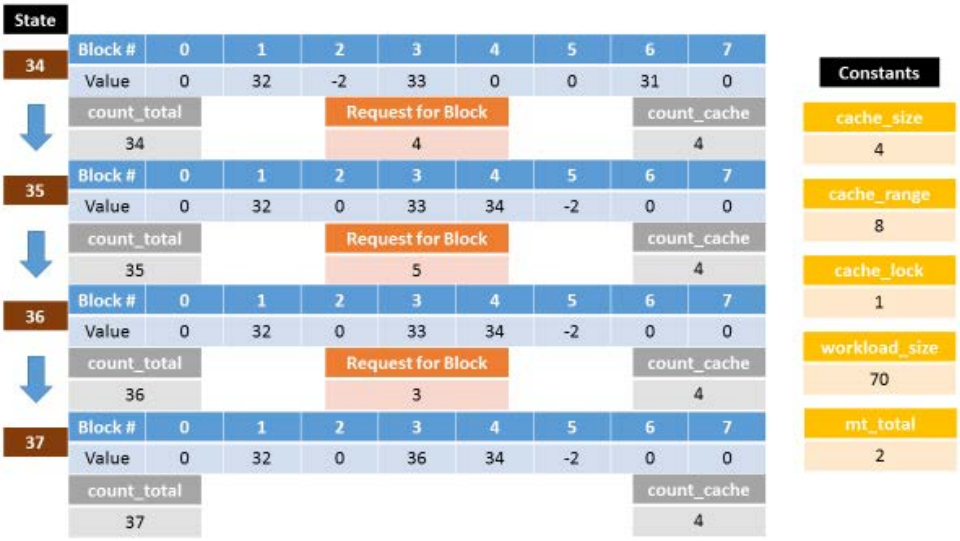

Figure 4.3.4. L2 Cache Array Design in Simulator and Demonstration Part 4

The last example shown in Figure 4.3.4 shows how cache is dynamically changed. As can see in state 34, block 2 is locked with the value of -2. The workload_size, which informs us of how many block requests will be made to the cache, is divided by the number of MTs (mt_total). This value when compared to count_total tells us when we need to replace the blocks locked. As can be seen in state 35, 2 is no longer locked and instead 5 becomes locked. In the case that the new value to be locked is already in the cache (is active) the count_cache is decremented by 1, and the block to be locked gains a negative number.

4.4 Set Associativity

In our tests, we used three different types of L2 cache associativity: 8-way, 16-way and full set associativity. 8-way set associativity was chosen as the baseline for most experiments, given its common use in real world systems. A second integer array is used during this process to retain what blocks are stored in what sets. The size of this array is the size of the cache. When a block is to be stored in this array, we first divide the block number down it down until the remainder is less than the size of the associative set. In the case of 8-way associativity, this would be from 0 – 7. A second

process then goes through and finds if a slot is open by repeatedly adding the associative set size to the remainder and checking that specific block. If all blocks are full, it then checks which is currently next up to be replaced then proceeds evict it from both integer arrays while implementing the new block.

A. Set 0	A. Set 1	A. Set 2	A. Set 3	A. Set 4	A. Set 5	A. Set 6	A. Set 7
0	1	2	3	4	5	6	7
7	8	9	10	11	12	13	14
15	16	17	18	19	20	21	22
...	...	...	...	...	...	...	...
232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247
248	249	250	251	252	253	254	255

Figure 4.4.1. L2 Cache Array Set Associativity Example

Figure 4.4.1 shows an example of how the set associativity is organized. The top bar indicates each associative set and the values below are the block numbers with that set. As can be seen, in 8-way associative set we tag 0, 7, 15, etc. all the way up to 248 together out of a possible range of 256 unique blocks. Given a block value of 252, it would be divided by our associative set size (8) until we obtain the remainder (4). This indicates that this block may be placed within associative set number 4. The total number of blocks retained by the set is dependent upon the size of the cache. Given a size of 256 with 8-way associativity, each set has 32 blocks it may retain at one time.

When cache locking is implemented, we victimize a percentage of each sets blocks we have available. For example, when implementing 25% cache locking we would have 24 blocks available for the cache replacement algorithm and 8 blocks locked.

4.5 Miss Table (MT)

The MTs are used to find the blocks that are most missed during the run-time of a process. In the case where four MTs are used, each keeps track of the cache miss count during its corresponding quarterly process run-time. All these MTs together show the total times each block is missed during the entire run-time. The information from MTs is then utilized in the following runs with the implementation of a unique cache locking mechanism. In our simulations, we tried different number of MTs: 4, 8, and 12.

The MTs were created using a 2D integer array. The first value of this array indicates the MT being populated, the second value indicates what block was missed, and the value stored at that location is the number of misses that occurred. We select what MT to receive the increments using the method described in section 4.3, where we have the workload_size divided by the number of MTs to obtain the coverage area of each MT. Once count_total reaches this number, it increments to the next MT.

4.6 Miss Ratio

The miss ratio in the L2 cache was simulated by adding a *chance-to-miss multiplier* to the total size of L2 cache. The resulting number was set as the upper bound for the random number generator (described in Section IV.A), and the lower bound was set to zero. For example, with this chance-to-miss multiplier value of 100%, the upper bound of the random number generator was set to 200% of the L2 cache size (100% L2 cache size plus 100% chance-to-miss-multiplier), which gives blocks a 50% chance to exist in the L2 cache.

4.7 Workloads

Instead of running actual processes in our simulator, we generated a list of

random numbers, as described in section 4.6. The size of the workload was typically varied from 20,000 to 100,000, with an increment of 20,000. To ensure accurate measurements, the number of simulations for each test was set to 10,000.

4.8 Cache Size

In our tests, we used four different L2 cache sizes as to be available for a process: 128, 256, 512, and 1024 block frames. This value relates to the cache_size as discussed earlier.

4.8 Cache Partition

The size of the locked cache partition (in L2 cache) was set to 25% of the whole L2 cache size for most of the experiments. When the set associativity was applied, each set had a corresponding number of cache blocks removed from the control of the replacement algorithm. For example, in the case of 8-way set associativity with 25% locked partition, 2 cache blocks were set aside for cache locking from every associative set.

CHAPTER 5

RESULTS

5.1 Introduction

After running our experiments, we compared the experimental results of the three locking schemes against the non-locking scheme. We found that while all locking schemes provide benefits, their individual performance varies under different conditions.

5.2 Simulations and Results

First, we evaluated how the locking schemes perform in 8-way set associativity. We found that their performance depends on the miss ratio (controlled by the value of the chance-to-miss multiplier described in chapter 4) and the workload size. As these values increased, the performance of each locking scheme decreased, as shown in Figs. 5.1–5.3.

Quarter locking (using 4 MTs) was the exception to this as it showed that its potential peaked at the multiplier value 40%. Despite this, the quarter locking scheme consistently outperformed the full locking scheme. Combining both quarter and full locking resulted in something half-way between the two in percent improvement. We also found that full locking quickly decreased in effectiveness as the chance-to-miss multiplier increased; dropping almost 2% in percent improvement in the worst case scenario. Quarter locking suffered a little more than 0.5% decrease in percent improvement in its worst case scenario.

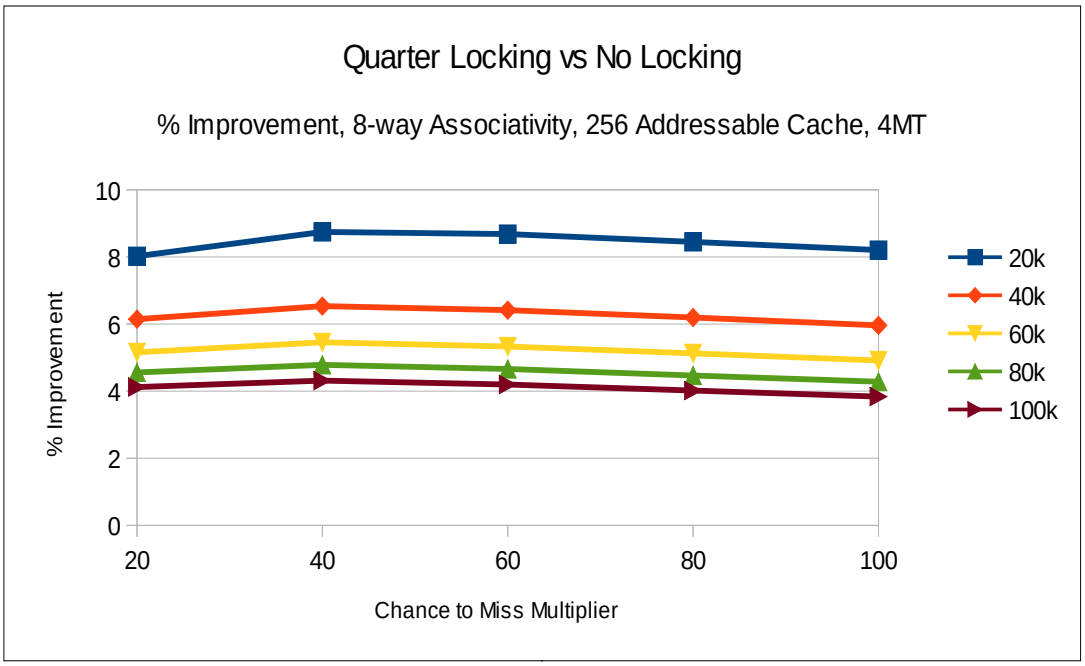

Figure 5.2.1. Quarter Locking vs No Locking

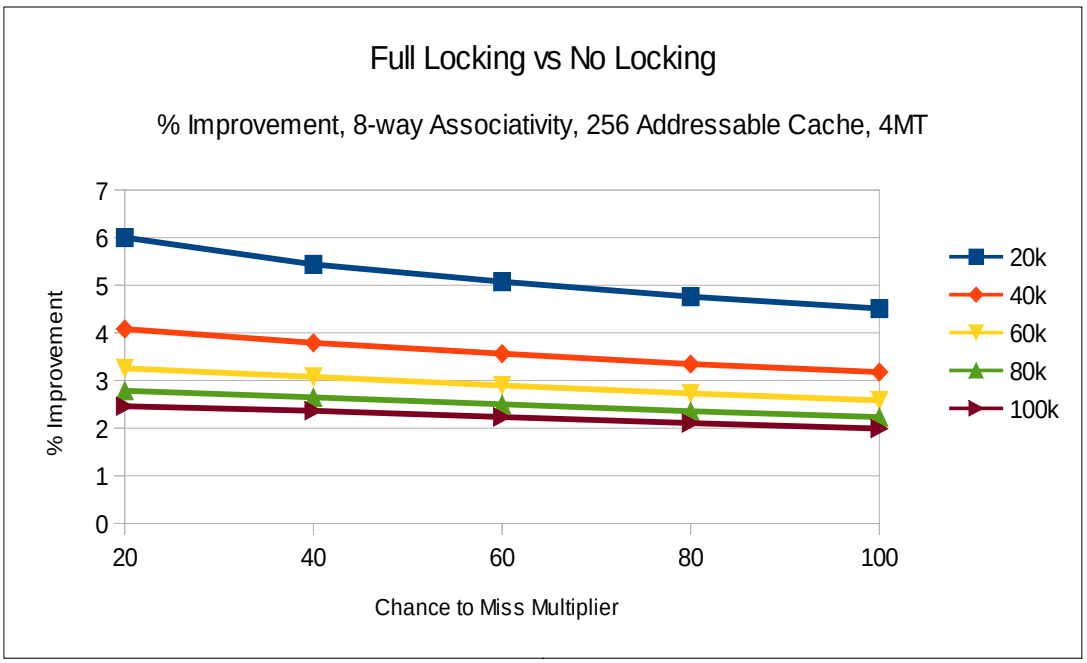

Figure 5.2.2. Full Locking vs No Locking

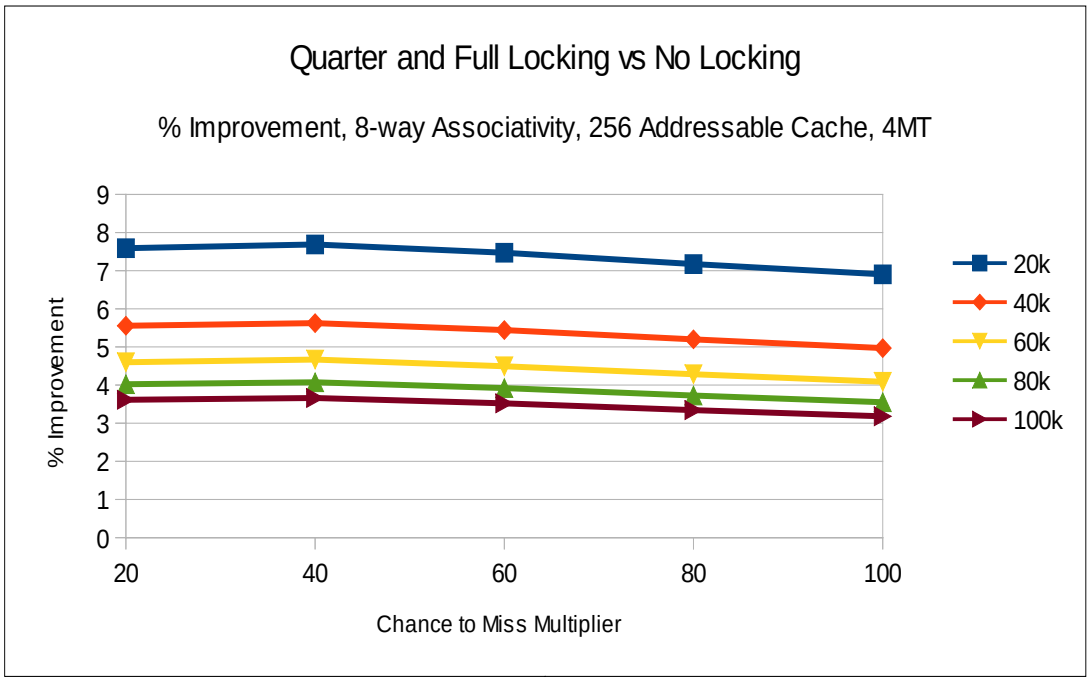

Figure 5.2.3. Quarter and Full Locking vs No Locking

Given the consistent improvement found with 4 MTs, we decided to test the effects of doubling or tripling the number of MTs used. Fig. 5.2.4 shows a slight improvement by increasing the number of MTs. Going from 4 MTs to 8 MTs, we gain about 1.5%; but going from 8 MTs to 12 MTs, we gain only half of that.

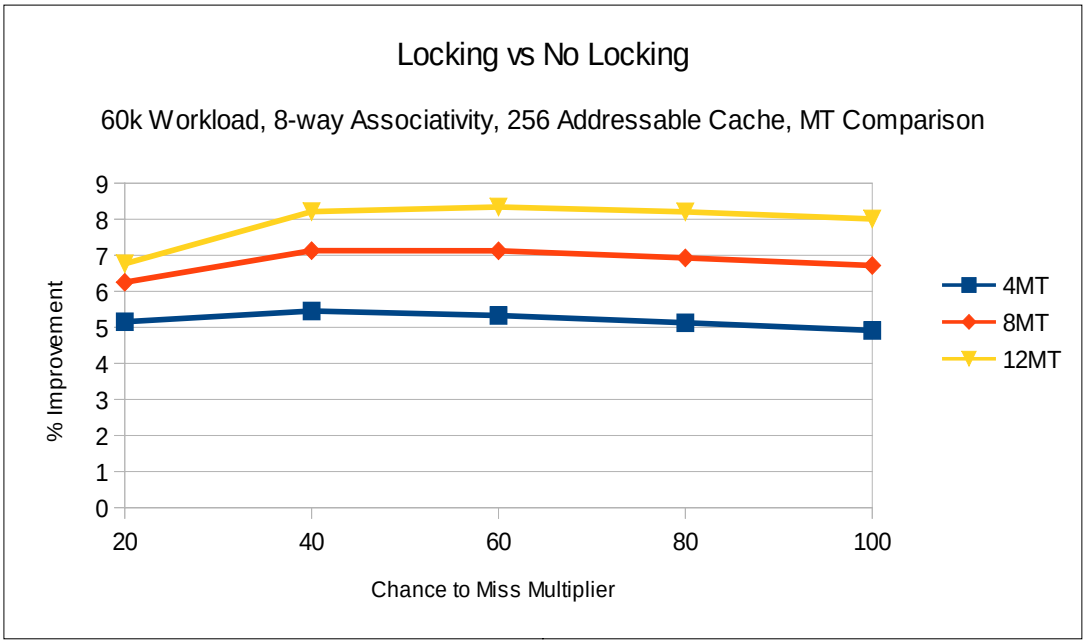

Figure 5.2.4. Locking vs No Locking, MT Comparison, 8-way Associativity

Next, we tested the effects of set associativity. We tried locking with 16-way associativity and full associativity. Combinations of different numbers of MTs and associative set sizes were also tested to see if any benefit obtained from one would be added with another.

Figs. 5.2.5 and 5.2.6 show that by increasing the set associativity there is little performance improvement. When the chance-to-miss multiplier is 40%, fully associative cache had a 5.8% improvement for quarter locking, while 16-way associativity had 5.5% improvement. From this, we can conclude that a larger set associativity has little effect on the percent improvement of cache locking over no locking.

In comparing a combination of 16-way set associativity and increased number of MTs, we found a greater increase in performance. With 16-way set associativity we achieved an 8.5% improvement over no locking with 12 MTs, as shown in Fig. 5.2.7. Under full associativity this number further increased to 8.9%, as shown in Fig. 5.2.8. These results indicate the additive effect in the improvement from both the increase in the number of MTs and the larger associative set size.

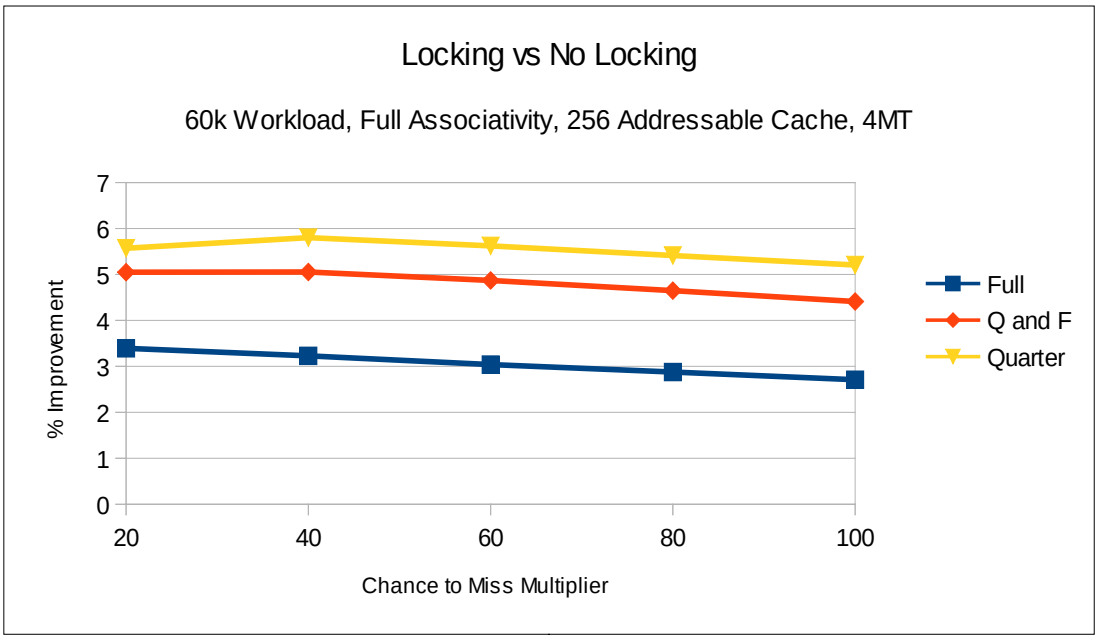

Figure 5.2.5. Locking vs No Locking, Full Associativity

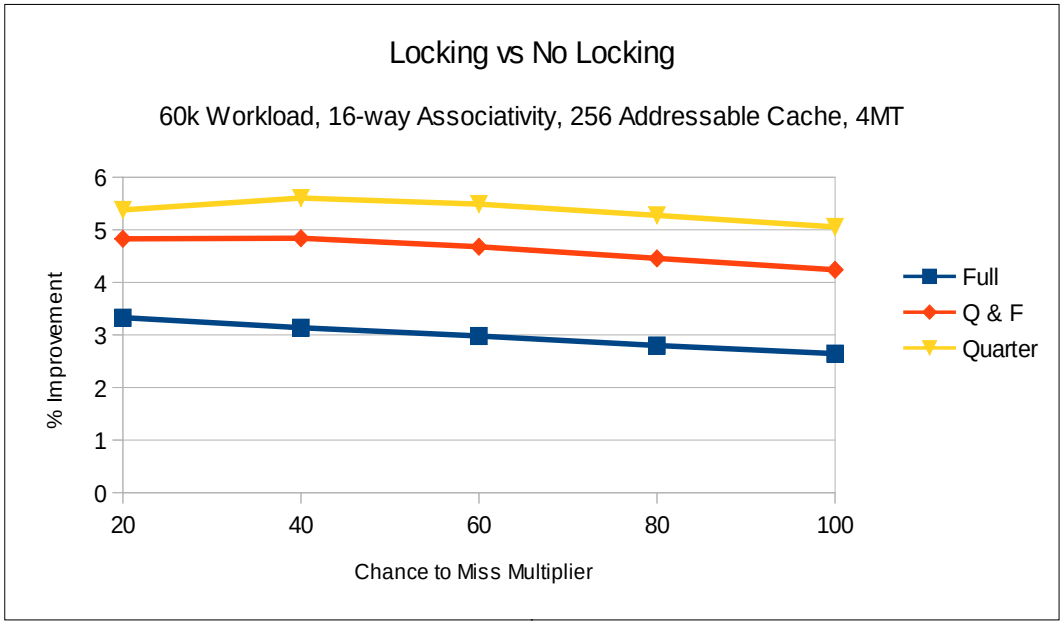

Figure 5.2.6. Locking vs No Locking, 16-way Associativity

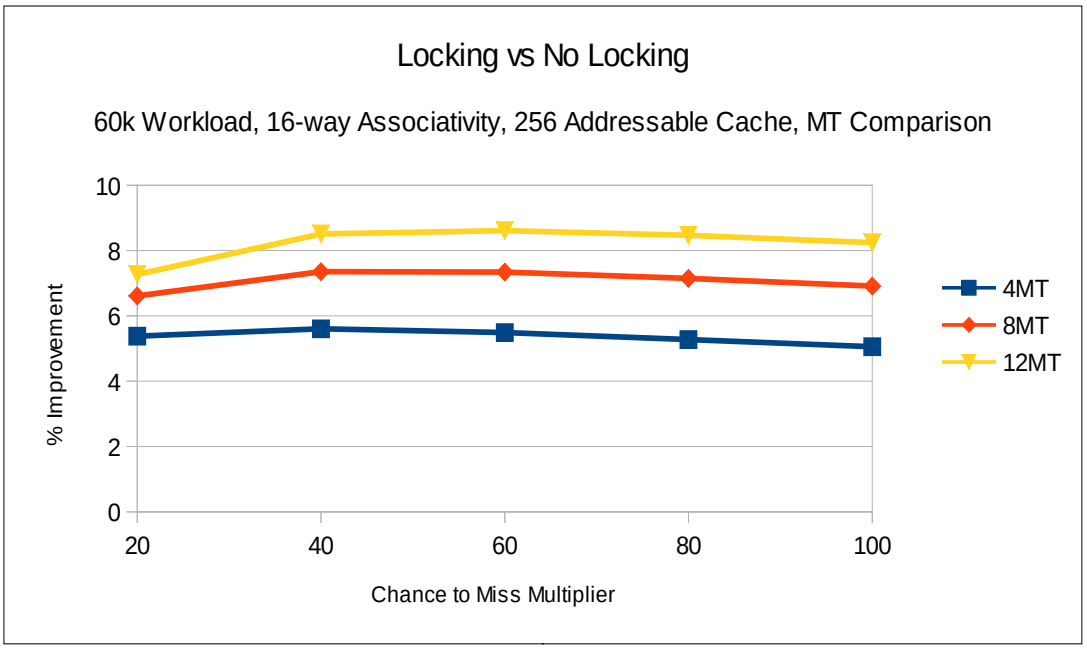

Figure 5.2.7. Locking vs No Locking, MT Comparison, 16-way Associativity

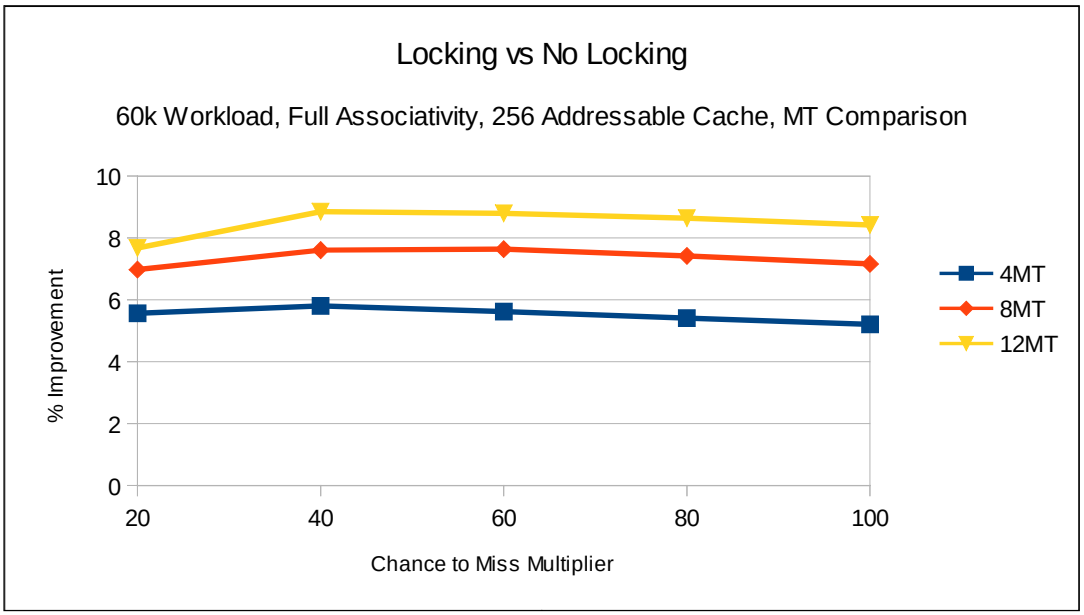

Figure 5.2.8. Locking vs No Locking, MT comparison, Full Associativity

We next varied the size of the locked partition. We doubled the locked partition from 25% to 50%, and also halved it to 12.5%. As shown in Fig. 5.2.9, while a higher percent improvement may be obtained by using 50% cache locked partition; this improvement quickly drops with a higher miss ratio. The 12.5% cache locked partition demonstrated a smaller but more consistent performance throughout all the miss ratios.

Fig. 5.2.10 shows the case of full locking, which has a similar pattern as quarter locking with respect to the locked cache partition size.

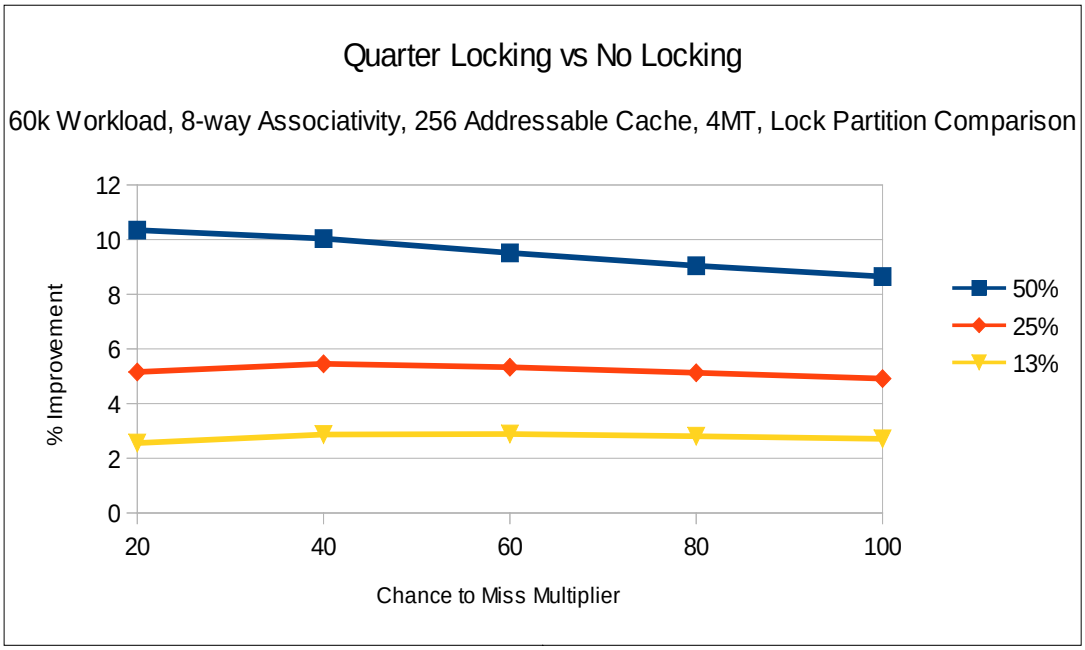

Figure 5.2.9. Quater Locking vs No Locking, lock partition comparison

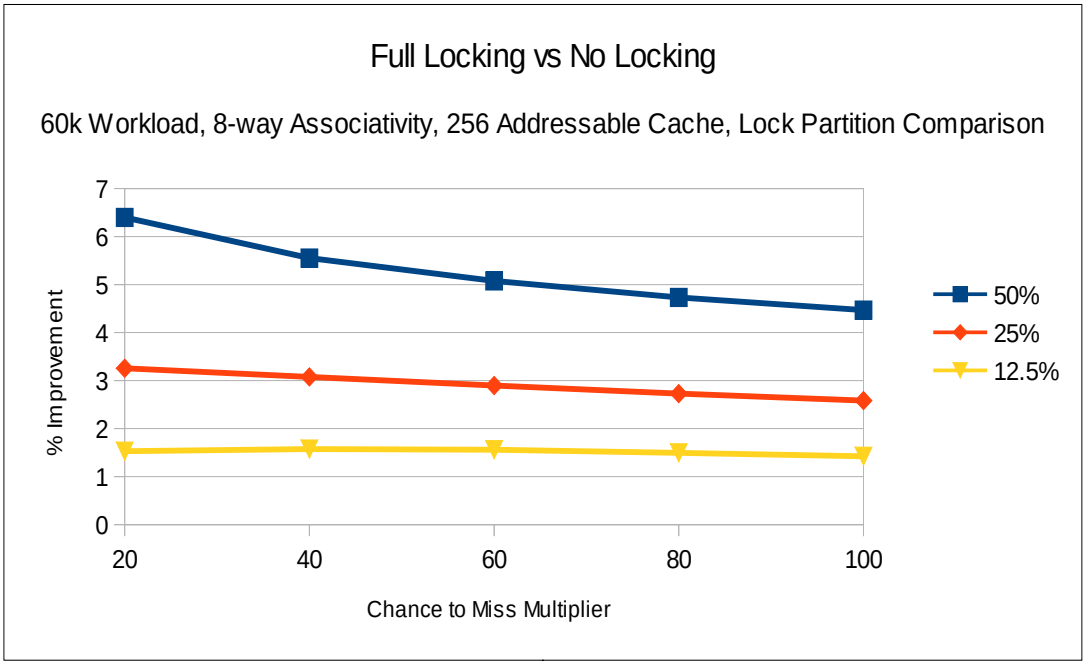

Figure 5.2.10. Full Locking vs No Locking, lock partition comparison

Our next series of experiments tested how well the cache locking schemes perform when the L2 cache size is reduced or increased. Figs. 5.2.11 and 5.2.12 shows that by reducing the number of cache block frames from 256 to 128, we

obtained a decreased percent improvement over no locking. When we increased it to 512 cache block frames, we obtained about a 2% improvement on top of the improvement obtained from the case of 256 cache block frames, as shown in Fig. 5.2.13. Fig. 5.2.14 shows that further increase of the L2 cache size to1024 block frames results in a better performance. So, we can claim our locking is scalable with the available L2 cache memory size for a process.

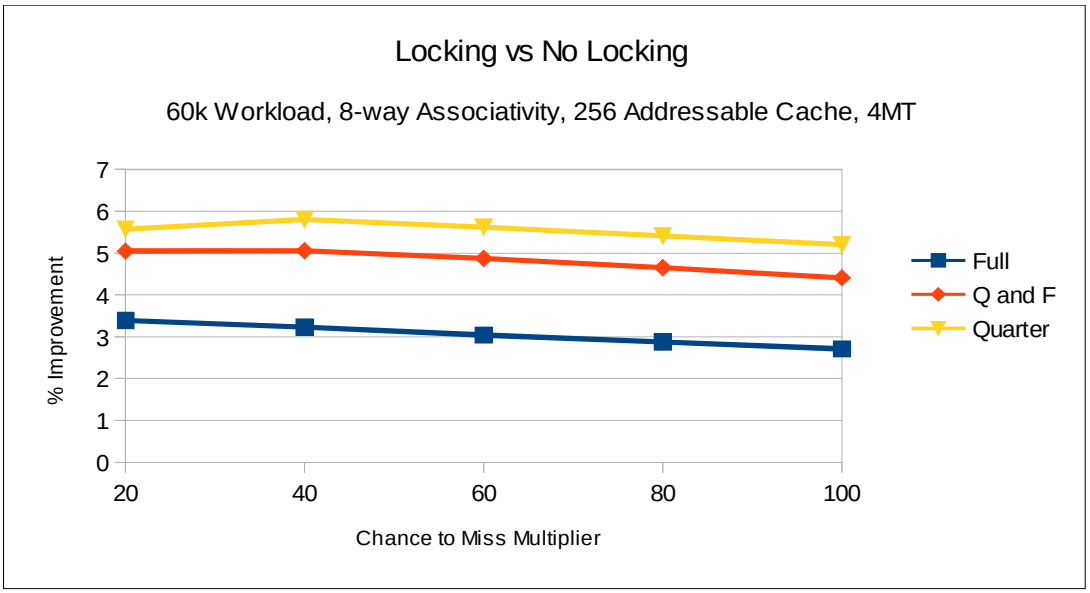

Figure 5.2.11. Locking vs No Locking, 256 addressable cache

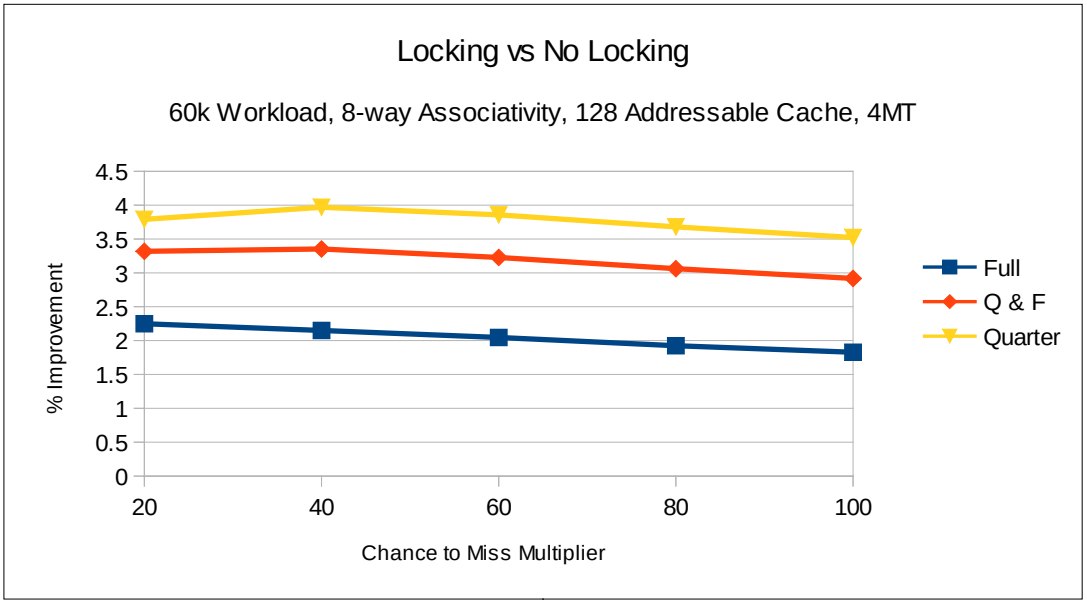

Figure 5.2.12. Locking vs No Locking, 128 addressable cache blocks

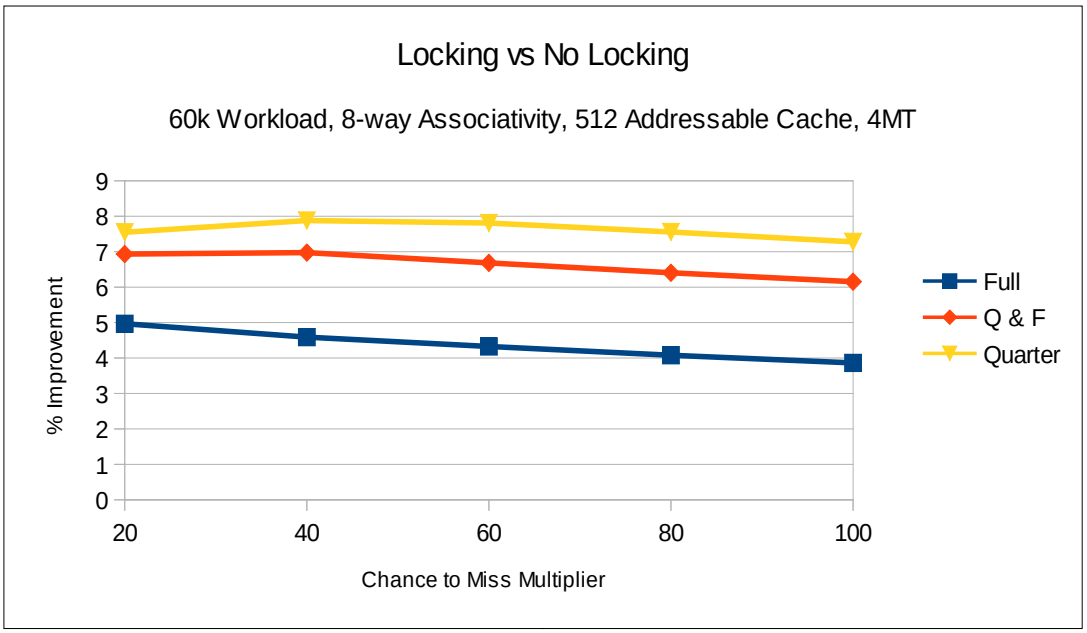

Figure 5.2.13. Locking vs No Locking, 512 addressable cache blocks

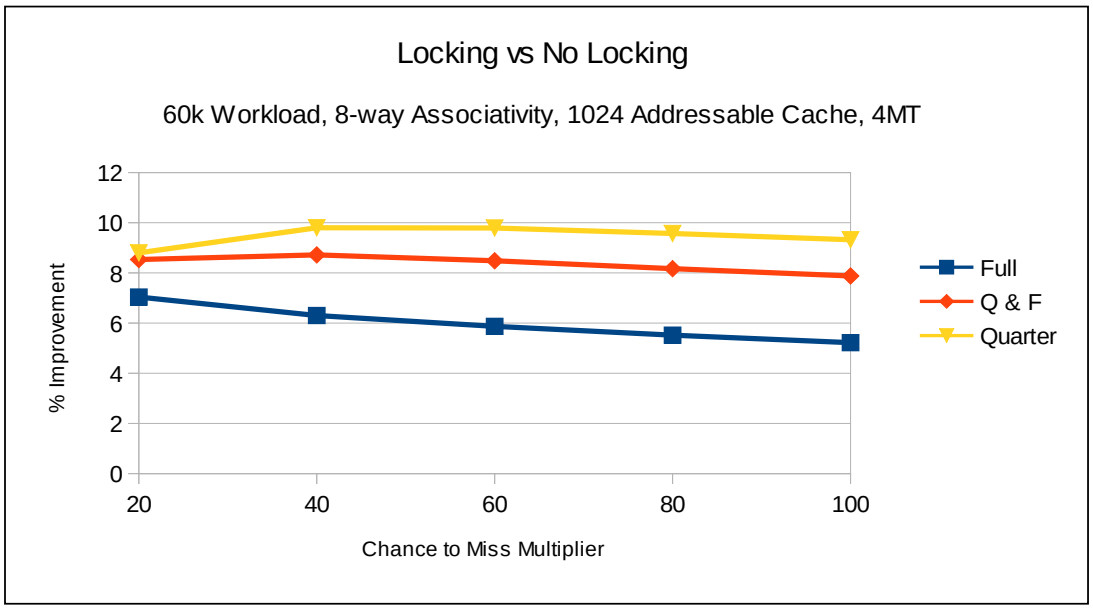

Figure 5.2.14. Locking vs No Locking, 1024 addressable cache.

In our last experiment, we increased the chance-to-miss multiplier up to 250%. This allowed us to see how a workload of bigger size than the cache would perform with our locking schemes. Fig. 5.2.15 shows that the performance greatly decreases at higher levels of miss ratio, with quarter locking still giving the best percent improvement.

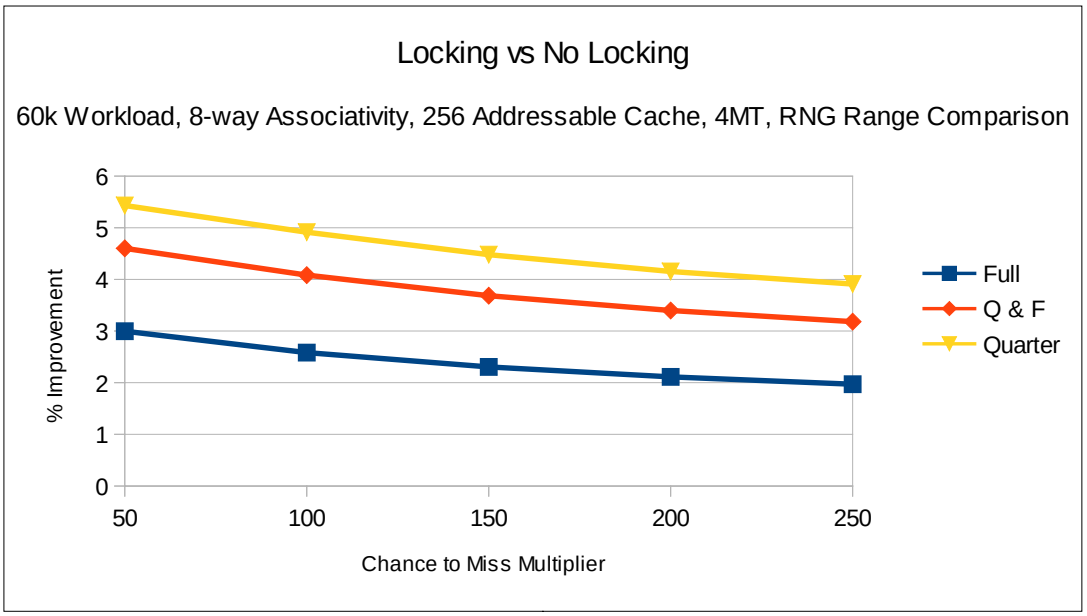

Figure 5.2.15. Locking vs No Locking, 50 to 250 chance-to-miss multiplier

CHAPTER 6

RELATED WORK

Previous researches explored the potential benefits of cache locking. It has been reported that cache locking may increase not only the speed at which a system performs, but its efficiency as well. In [1], the increased efficiency comes from the reduction in the number of cache misses accrued from running an optimized process by utilizing MT based cache locking.

In [9], randomly selected blocks are locked in the cache memory. However, some of the randomly select blocks may not be used frequently, whereas frequently used blocks would be kept in the cache memory anyway by the cache replacement algorithm.

In [6], they found that locking the blocks in the L1 I-cache results in a better worst-case execution time, even though accurate estimation of the run-time of a process is not feasible.

Complex processes with branching may lead to varying results with cache locking. A branch prediction algorithm has shown some success in improving the chance of accurately selecting the blocks to be locked during execution time [7]. Another method is locking blocks based on which region in the process is actively being used, and it resulted in an increase in the system's dynamic performance [8].

CHAPTER 7

CONCLUSION AND FUTURE WORKS

7.1 Conclusion

An optimized cache memory management technique results in the improvement of a process's performance. Cache locking is one such technique that has been found to be beneficial [1, 2, 5].

In this research, we proposed a new cache locking technique which utilizes multiple miss tables (MTs). Each MT keeps track of all the misses in the respective portion of the run-time of a process; and the total MT, which is a combination of all these MTs, shows all the misses during the whole run-time.

These MTs allow for the cache memory management system to select only the most missed blocks to be locked during each portion of the run-time. This resulted in a noticeable increase in percent improvement against the case using only the LRU replacement algorithm. It has been found that this performance gain may be further increased by increasing the number of MTs and the size of locked cache partition.

We also explored dividing the cache memory into three partitions. One partition was devoted to the LRU cache replacement algorithm, and the other two to cache locking. Among these two locked partitions, one uses the information of the individual MTs, and the other one uses the information of the total MT. We found that this method performed superior to full locking, but it was inferior to the case of using only individual MTs.

7.2 Future Works

Research into L2 cache locking has consistently shown that it is beneficial in the reduction of cache misses. Stressing such a locking scheme under various real world conditions such as branching should be conducted to find how well it would perform when integrated to real world systems.

Our research was only on the feasibility of dynamically locking cache with the use of MTs. As such, we designed and used a simple simulator in order to investigate how it would perform. We did not include any of the advanced cache replacement techniques in modern computers nor did we consider the operation of real-time programs that may branch.

REFERENCES

[1] A. Asaduzzaman, F. N. Sibai, and M. 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, vol. 56, no. 4-6, pp. 151–162, 2010.

[2] V. Suhendra and T. Mitra, “Exploring Locking & Partitioning for Predictable Shared Caches on

Multi-cores,” Proc. of the 45th annual Design Automation Conference, 2008, pp. 300–303.

[3] J. Hennessy and D. Patterson, *Computer Architecture*, 5th Edition, 2012.

[4] P. Denning, “Virtual Memory,” ACM Computing Surveys, vol. 28, no. 1, 1996.

[5] I. Puaut and D. Decotigny, “Low-complexity Algorithms for Static Cache Locking in

Multitasking Hard Real-time Systems,” Proc. of IEEE Real-Time Systems Symp., pp. 114–123,

2002.

[6] T. Liu, M. Li, and C. Xue, “Minimizing WCET for Real-Time Embedded Systems via Static

Instruction Cache Locking,” Proc. of Real-Time and Embedded Technology and Applications

Symposium, 2009, pp. 35–44.

[7] K. Qiu, M. Zhao, C. Xue, and A. Orailoglu, “Branch Prediction Directed Dynamic Instruction

Cache Locking for Embedded Systems,” Proc. of Int’l Conf. on Embedded and Real-Time

Computing Systems and Applications, 2013, pp. 209–216.

[8] A. Arnaud and I. Puaut, “Dynamic Instruction Cache Locking in Hard Real-Time Systems,”

Proc. of Int’l Conf. on Real-Time and Network Systems, 2006.

[9] A. Asaduzzaman, “An Effective Level-1 Cache Locking Strategy for Energy-Efficient

Real-Time Multicore Systems,” Proc. of Int’l Conf. on Computer and Information Technology,

2011, pp. 18–23