A New Approach To Set-based Dynamic Cache Partitioning on Chip Multiprocessors

by

Esen Varol

Submitted to the Institute of Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

in

Computer Engineering

Yeditepe University

2014

A New Approach To Set-based Dynamic Cache Partitioning on Chip Multiprocessors

APPROVED BY:

Assist. Prof. Gürhan KÜÇÜK (Supervisor)	....
Assist. Prof. Tacha SERIF	....
Prof. Dr. Nejat YUMUŞAK	....

DATE OF APPROVAL: ...../...../2014

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Assistant Professor Gürhan Küçük for the continuous support of my M.Sc. study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my M.Sc.study.

Special thanks go to my co-partner Engin Ören, who gave me his unconditional support.

I would like to thank my family: my parents Nedret and Nihat Varol, for giving birth to me at the first place and supporting me spiritually throughout my life.

Finally, and most importantly, I would like to thank my wife Özlem. Her support, encouragement, quiet patience and unwavering love were great.

To my lovely wife Özlem and my deary daughter Defne.

ABSTRACT

A NEW APPROACH TO SET-BASED DYNAMIC CACHE PARTITIONING ON CHIP MULTIPROCESSORS

Modern processors contain multiple cores which enables them to concurrently execute multiple applications on a single chip in parallel. As the number of cores on a chip increases, the pressure on the memory system to sustain the memory requirements of all the concurrently executing threads increases. Worst of all, today’s processor architectures bring many cores next to each other and hope that the applications running on these cores are going to share the last level cache without any problem. As a result, applications, which have no clue that there are other neighbors competing for the same resources, think that each of them has a dedicated cache and start a competition of stealing cache lines from each other without even knowing it. In such cases, an application with a large memory foot print may spoil the cache and, suddenly, this may render the last level cache in a position bringing more harm (drop of performance, unnecessary power and energy dissipation) than any good. So, one of the keys to obtaining high performance from multicore architectures is to manage the Last Level Cache (LLC) efficiently.

This new approach calculates the efficiency of each application with the help of application-based runtime statistics collected by hardware counters. According to these statistics, we classify the threads that are executed concurrently. At the end, we try assigning Last Level Cache partitions to running threads considering their instant behavior detected by the Classifier circuit. Consequently, the system aims to improve the performance of individual applications and total system throughput by giving more cache sets to the applications with more throughput potentials when they receive more cache resources. The secondary aim of the proposed method is its scalable design for many-core architectures.

ÖZET

ÇOKLU MİKROİŞLEMCİLERDE SET-BAZLI DİNAMİK ÖNBELLEK PAYLAŞIMINA YENİ BİR YAKLAŞIM

Modern işlemciler, tek bir çip üzerinde paralel olarak aynı anda birden fazla uygulama çalıştırmasına olanak sağlayan çoklu çekirdek içerir. Bir çip içerisindeki çekirdek sayısı arttıkça, aynı anda çalışan tüm uygulamaların bellek gereksinimlerini sürdürmek için bellek sistemi üzerindeki baskı artar. Hepsinden kötüsü, günümüz işlemci mimarileri birçok çekirdeği yanyana getirip, bu çekirdekler üzerinde çalışan uygulamaların herhangi bir sorun olmadan son seviye önbelleği paylaşmasını ummaktadır. Sonuç olarak, bu uygulamaların aynı bellek kaynakları için rekabet eden diğer komşuları olduğununa dair hiçbir ipuçları olmadığı gibi, bu uygulamaların her biri özel bir önbelleğe sahip olduğunu da düşünmektedir ve hatta bilmeden birbirinden önbellek hatları çalmak için rekabet içine girmektedirler. Bu gibi durumlarda, bir uygulama geniş bir bellek ayak izi ile önbelleği yağmalayabilir, bu durum aniden son seviye önbelleği, faydadan çok daha zararlı (performans düşüşü, gereksiz güç ve enerji yayılımı) bir pozisyona getirebilir. Yani, çok çekirdekli mimarilerden yüksek performans elde etmek için gerekli anahtarlardan biri, son seviye önbelleği verimli olarak yönetmektir.

Bu yeni yaklaşım, donanım sayaçları tarafından toplanan uygulama tabanlı çalışma zamanı istatistikleri yardımı ile her bir uygulamanın verimini hesaplar. Bu istatistiklere göre, biz aynı anda çalışan uygulamaları sınıflandırmaktayız. Sonunda, uygulamaların sınıflandırıcı devresi tarafından tespit edilen anlık davranışlarını dikkate alarak, aynı anda çalışan bu uygulamalara Son Seviye Önbellek bölümlerini paylaştırmayı denemekteyiz. Sonuç olarak bu çalışma ile, daha fazla önbellek kaynağı aldığında, daha fazla iş çıkarma potansiyeline sahip uygulamalara daha fazla önbellek seti vererek, sistemin bireysel uygulamaların toplam verimi ve performansını geliştirmesi amaçlamaktadır. Önerilen yöntemin ikincil amacı, çok çekirdekli mimariler için ölçeklenebilir bir tasarımdır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT v
ÖZET vi
LIST OF FIGURES viii
LIST OF TABLES xii
LIST OF SYMBOLS xiii
1. INTRODUCTION 1
    1.1. CHIP MULTIPROCESSORS 1
    1.2. THE APPLICATION PARALLELISM LANDSCAPE 7
    1.3. CACHE ORGANIZATION 10
    1.4. MOTIVATION 15
2. LITERATURE SURVEY 20
3. SET-BASED DYNAMIC CACHE PARTITIONING ON CMPs 29
    3.1. CACHE STATICTICS 30
    3.2. CLASSIFIER ALGORITHM 32
    3.3. PARTITIONER ALGORITHM 37
    3.4. HARDWARE IMPLEMENTATION AND COMPLEXITY OVERHEAD 46
4. TEST ENVIRONMENT 50
    4.1. MACSIM SIMULATOR 50
    4.2. BENCHMARKS 53
    4.3. TRACE GENERATION 54
5. EXPERIMENTAL RESULTS 59
6. CONCLUSIONS AND FUTURE WORK 66
REFERENCES 68

LIST OF FIGURES

Figure 1.1. Floorplan for the six-issue dynamic superscalar microprocessor 1

Figure 1.2. Intel processor normalized performance per cycle over time 2

Figure 1.3. Superscalar architecture showing multiple executions 3

Figure 1.4. Intel processor power over time 4

Figure 1.5. CMPs Architecture 5

Figure 1.6. Homogeneous vs. Heterogeneous CMPs 6

Figure 1.7. A summary of the various “ranges” of parallelism 8

Figure 1.8. Cache hit 11

Figure 1.9. Cache miss 12

Figure 1.10. Direct Mapping Method 13

Figure 1.11. Fully associative cache mapping 13

Figure 1.12. Set associative cache mapping 14

Figure 1.13. Cache Statistics for “gcc” 17

Figure 1.14. Cache Statistics for “lbm”. 17

Figure 1.15. Cache Statistics for “lbm” and “gcc” Mixture 18

Figure 1.16. Cache Statistics for “mcf” 19

Figure 2.1. Hardware implementation of Utility-Based Cache Partitioning 22

Figure 2.2. Vantage managed-unmanaged region division 27

Figure 2.3. HitMaximization: Gains provided by PriSM over way partitioning 28

Figure 2.4. Performance of Vantage and PriSM in 4 and 16 core machines 28

Figure 3.1. Our proposed design cache organization 29

Figure 3.2. Logic cache partitioning of our design 30

Figure 3.3. Defining Global Variables 35

Figure 3.4. Classify() method - 1 36

Figure 3.5. Classify() method - 2 37

Figure 3.6. Steps of partitioning algorithm 39

Figure 3.7. Conflict Resolution Algorithm 40

Figure 3.8. AssignPartition Algorithm - 1 41

Figure 3.9. AssignPartition Algorithm – 2 42

Figure 3.10. AssignPartition Algorithm – 3 43

Figure 3.11. AssignPartition Algorithm – 4 44

Figure 3.12. SearchConflictResolution Algorithm 45

Figure 3.13. ShiftConflictEntries Algorithm 46

Figure 3.14. The space requirements of the design variation with the number of cores 47

Figure 3.15. Hardware implementation of classifier 48

Figure 3.16. Hardware implementation of partitioner 49

Figure 4.1. Cache Structure of MACSIM 51

Figure 4.2. Memory System of MacSim 52

Figure 4.3. The Overview of MACSIM Simulator 55

Figure 4.4. Trace Generation Script 55

Figure 4.5. Trace.txt format 56

Figure 4.6. The Structure/Format for Encoding Instructions 58

Figure 5.1. Results of bzip2, milc, povray and gcc 59

Figure 5.2. Results of bzip2, milc, omnetpp and gcc 60

Figure 5.3. Results of bzip2, namd, povray and gcc 61

Figure 5.4. Results of bzip2, sjeng, lbm and astar 61

Figure 5.5. Results of libquantum, mcf, href264 and astar 62

Figure 5.6. Results of libquantum, milc, href264 and gcc 63

Figure 5.7. Results of lbm, gcc, povray and bzip2 63

Figure 5.8. Results of bzip2, milc, povray, gcc, omnetpp, namd, sjeng, href264 64

Figure 5.9. Whole Test Results...65

LIST OF TABLES

Table 1.1. Multi-core processor configuration 16

Table 1.2. Performance loss for Mixture “gcc” and “lbm” 18

Table 1.3. Performance loss for Mixture “gcc” and three “mcf”s 19

Table 3.1. Example of Partitioner Table...38

Table 3.2. Classifier cost...48

Table 5.1. Experimental Setup Table...59

Table 6.3. New distribution approach for future works...66

LIST OF SYMBOLS / ABBREVIATIONS

ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
I/O	Input Output
B/W	Band Width
CMP	Chip Multiprocessors
Dcache	Data Cache
ILP	Instruction Level Parallelism
IPC	Instruction Per Cycle
Icache	Instruction Cache
IQ	Issue Queue
L1	Level One
L2	Level Two
L3	Level Three
LRU	Least Recently Used
LSQ	Load/Store Queue
MRU	Most Recently Used
MLP	Memory Level Parallelism
O/S	Operating System
PL1C	Private Level One Cache
PL2C	Private Level Two Cache

ROB	Reorder Buffer
RF	Register File
SPEC	Standard Performance Evaluation Corporation
SRAM	Static Random Access Memory
TLP	Thread Level Parallelism
UMON	Utility Monitor
WIPC	Weighted IPC

1. INTRODUCTION

1.1. CHIP MULTIPROCESSORS

As the number of transistors on chip continues to increase there came numerous ways to better utilize the silicon on chip. The initial approach was the **superscalar** architecture (shown in Figure 1.1). Superscalar means executing multiple instructions at the same time. Superscalar processors were developed to execute multiple instructions from a single, conventional instruction stream on each cycle. These function by dynamically examining sets of instructions from the instruction stream to find ones capable of parallel execution on each cycle, and then executing them, often out-of-order with respect to the original sequence. This takes advantage of any parallelism that may exist among the numerous instructions that a processor executes, a concept known as **instruction-level parallelism** (ILP). Both pipelining and superscalar instruction issues have flourished because they allow instructions to execute more quickly while maintaining the key illusion for programmers that all instructions are actually being executed sequentially and in-order, instead of overlapped and out-of-order.

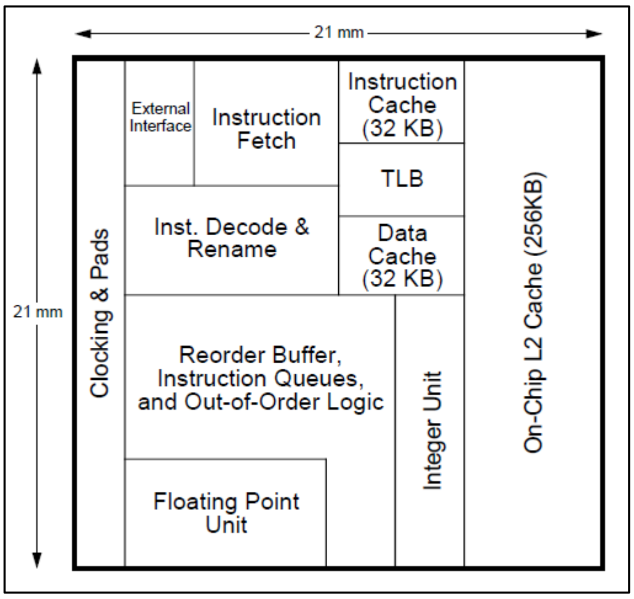


Figure 1.1 Floorplan for the six-issue dynamic superscalar Microprocessor [1]

Unfortunately, it is becoming increasingly difficult for processor designers to continue using these techniques to enhance the speed of modern processors. Typical instruction streams have only a limited amount of usable parallelism among instructions, so superscalar processors that can issue more than about four instructions per cycle achieve very little additional benefit on most applications. Figure 1.2 shows how effective real Intel processors have been at extracting instruction parallelism over time. There is a flat region before instruction-level parallelism was pursued intensely, then a steep rise as parallelism was utilized usefully, and followed by a tapering off in recent years as the available parallelism has become fully exploited. [2]

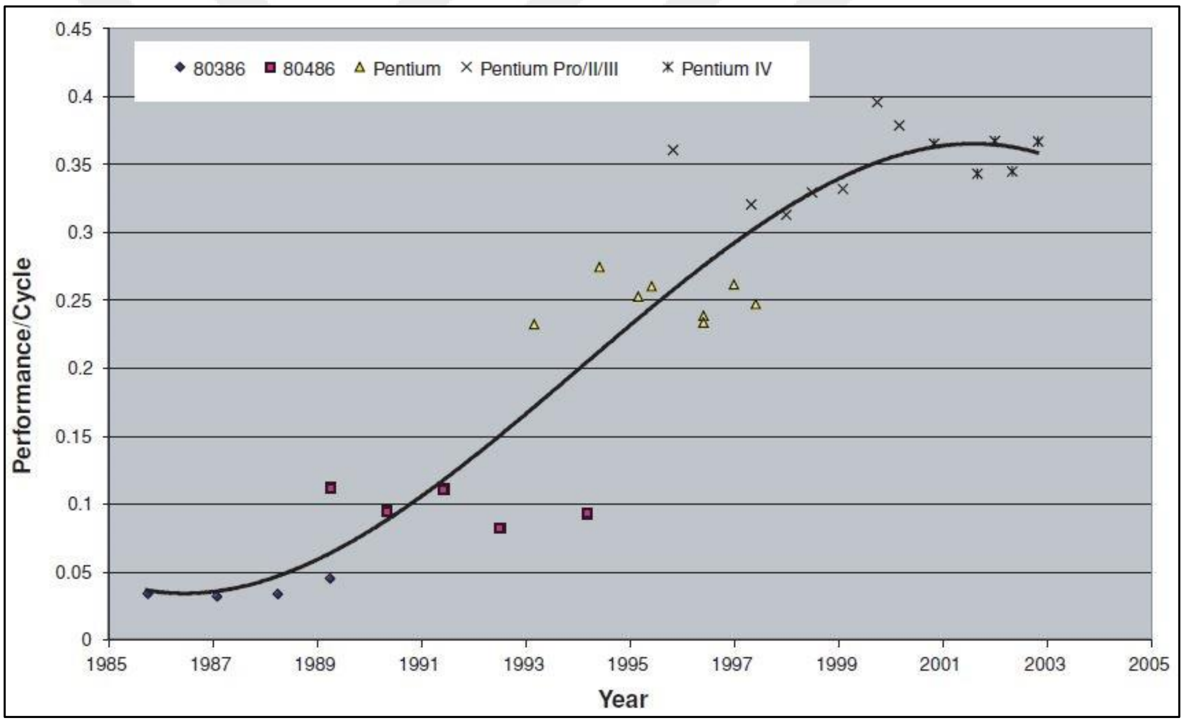


Figure 1.2. Intel processor normalized performance per cycle over time [2]

Complicating matters further, building superscalar (Figure 1.3.) processor cores that can exploit more than a few instructions per cycle becomes very expensive, because the complexity of all the additional logic required to find parallel instructions dynamically is approximately proportional to the square of the number of instructions that can be issued simultaneously. Similarly, pipelining past about 10–20 stages is difficult because each pipeline stage becomes too short to perform even a basic logic operation, such as adding two integers together, and subdividing circuitry beyond this point is very complex. In addition, the circuitry overhead from adding additional pipeline registers and bypass path multiplexers to the existing logic combines with performance losses from events that cause pipeline state to be flushed, primarily branches, to overwhelm any potential performance gain from deeper pipelining after about 30 stages or so. Further advances in both superscalar issue and pipelining are also limited by the fact that they require ever-larger number of transistors to be integrated into the high-speed central logic within each processor core—so many, in fact, that few companies can afford to hire enough engineers to design and verify these processor cores in reasonable amounts of time. These trends slowed the advance in processor performance, but mostly forced smaller vendors to forsake the high-end processor business, as they could no longer afford to compete effectively.

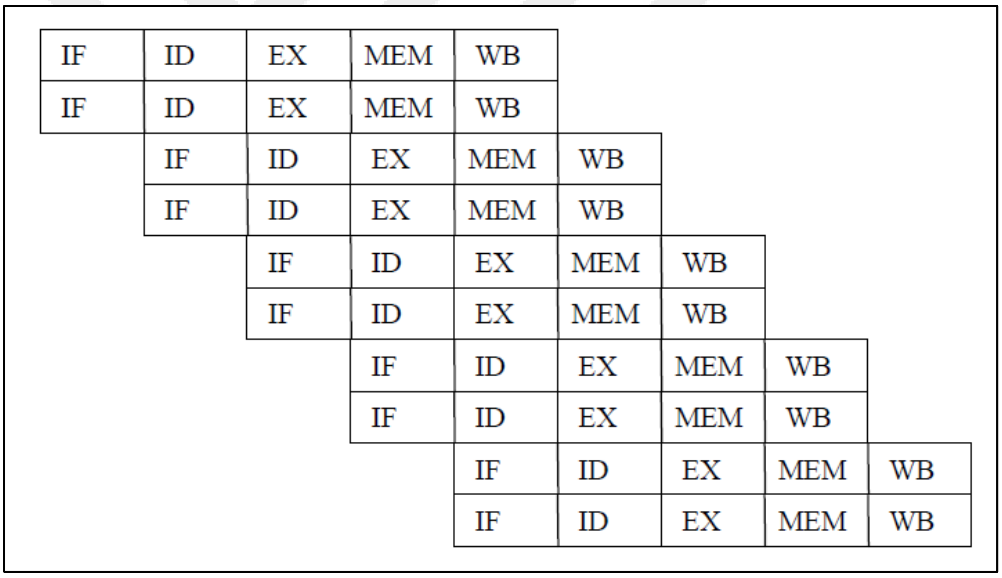


Figure 1.3. Superscalar architecture showing multiple executions

Today, however, progress in processor core development has slowed dramatically because of a simple physical limit: power. As processors were pipelined and made increasingly superscalar over the course of the past two decades, typical high-end microprocessor power went from less than a watt to over 100W. Even though each silicon process generation promised a reduction in power, as the ever-smaller transistors required less power to switch, this was only true in practice when existing designs were simply “shrunk” to use the new process technology. However, processor designers kept using more transistors in their cores to add pipelining and superscalar issue, and switching them at higher and higher frequencies, so the overall effect was that exponentially more power was required by each subsequent processor generation (as illustrated in Figure 1.4).

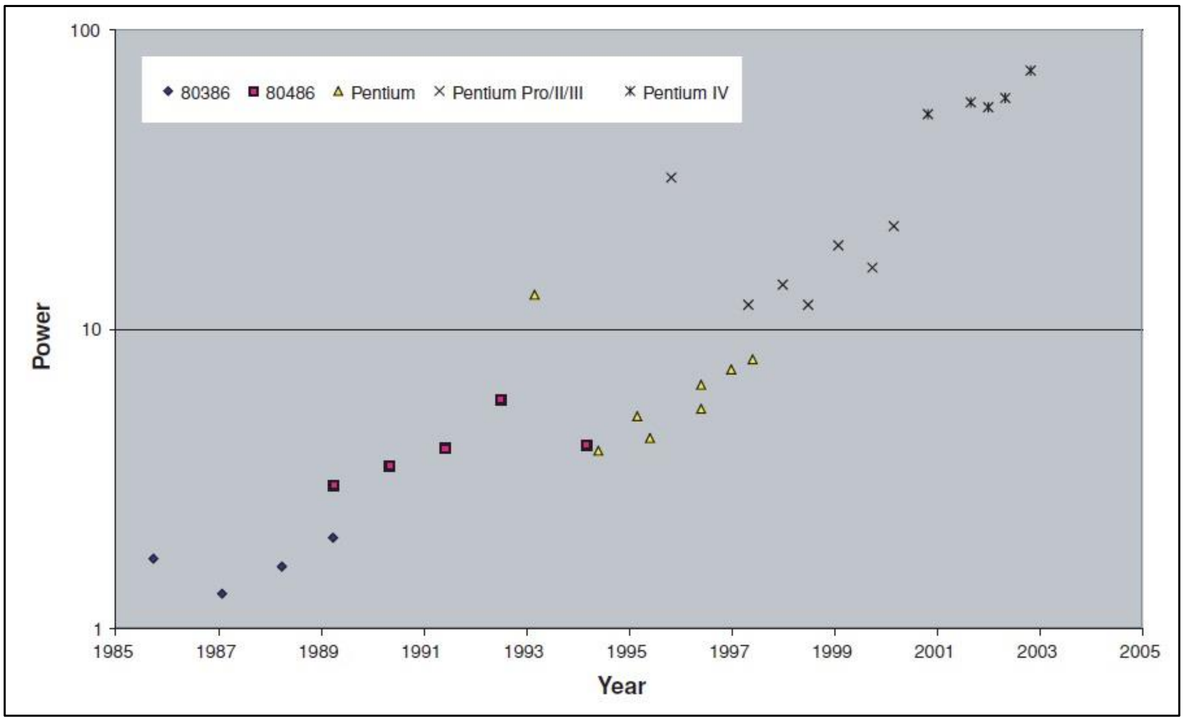


Figure 1.4. Intel processor power over time

Unfortunately, cooling technology does not scale exponentially nearly as easily. As a result, processors went from needing no heat sinks in the 1980s, to moderate-sized heat sinks in the 1990s, to today's monstrous heat sinks, often with one or more dedicated fans to increase airflow over the processor. If these trends were to continue, the next generation of microprocessors would require very exotic cooling solutions, such as dedicated water cooling, that are economically impractical in all but the most expensive systems.

The combination of finite instruction parallelism suitable for superscalar issue, practical limits to pipelining, and a "power ceiling" set by practical cooling limitations limits future speed increases within conventional processor cores to the basic Moore's law improvement rate of the underlying transistors. While larger cache memories will continue to improve performance somewhat, by speeding access to the single "memory" in the conventional model, the simple fact is that without more radical changes in processor design, one can expect that microprocessor performance increases will slow dramatically in the future, unless processor designers find new ways to effectively utilize the increasing transistor budgets in high-end silicon chips to improve performance in ways that minimize both additional power usage and design complexity.

These limits have combined to create a situation where ever-larger and faster uniprocessors are simply impossible to build. In response, processor manufacturers are now switching to a new microprocessor design paradigm: **the chip multiprocessor, or CMP**. A multi-core processor (known as a chip multiprocessor or CMP) is a single computing component with two or more independent actual central processing units (called "cores"), which are the units that read and execute program instructions. The instructions are ordinary CPU instructions such as add, move data, and branch, but the multiple cores can run multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing. Manufacturers typically integrate the cores onto a single integrated circuit die, or onto multiple dies in a single chip package. (as shown by Figure 1.5.)

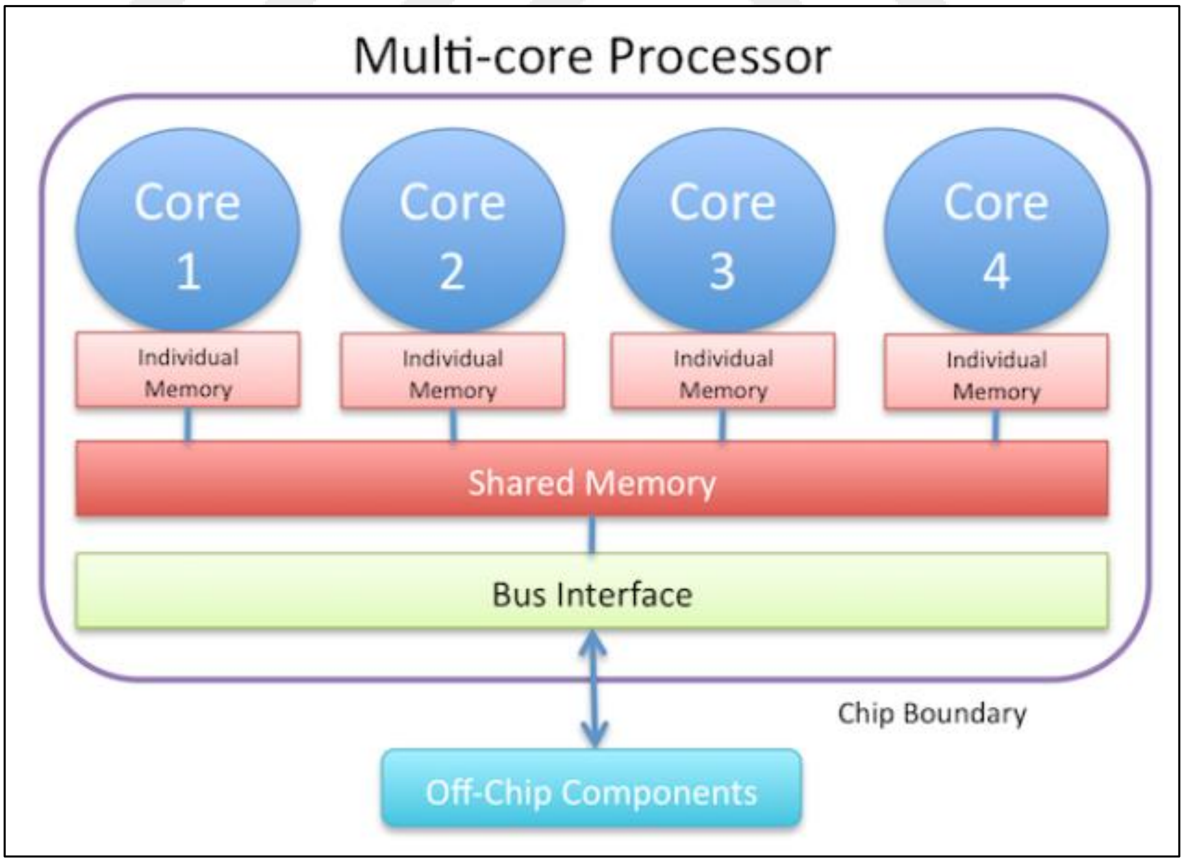


Figure 1.5. CMPs Architecture

Processors were originally developed with only one core. Multi-core processors were developed in the early 2000s by Intel, AMD and others. Multicore processors may have two cores (dual-core CPUs, for example AMD Phenom II X2 and Intel Core Duo), four cores (quad-core CPUs, for example AMD Phenom II X4, Intel's i5 and i7 processors), six cores (hexa-core CPUs, for example AMD Phenom II X6 and Intel Core i7 Extreme Edition 980X), eight cores (octo-core CPUs, for example Intel Xeon E7-2820 and AMD FX-8350), ten cores (for example, Intel Xeon E7-2850), or more. A multi-core processor implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared memory inter-core communication methods. Common network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar. As you see in the figure 1.6., Homogeneous multi-core systems include only identical cores, heterogeneous multi-core systems have cores that are not identical.

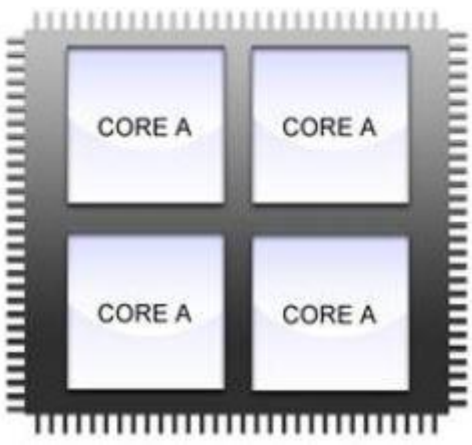
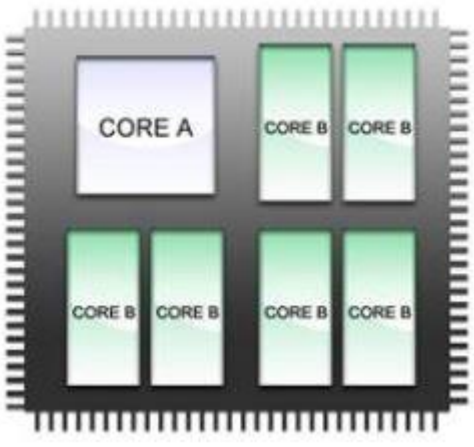
Homogeneous Multi-Core Processor Configuration	Heterogeneous Multi-Core Processor Configuration
	 CORE A CORE B CORE B CORE B CORE B CORE B CORE B
Multiple cores of the same type are implemented in one CPU.	Multiple cores of different types are implemented in one CPU.

Figure 1.6. Homogeneous vs. Heterogeneous CMPs

CMPs require a more modest engineering effort for each generation of processors, since each member of a family of processors just requires stamping down a number of copies of the core processor and then making some modifications to relatively slow logic connecting the processors to tailor the bandwidth and latency of the interconnect with the demands of the processor, but does not necessarily require a complete redesign of the high-speed processor pipeline logic. Moreover, unlike with conventional multiprocessors with one processor core per chip package, the system board design typically only needs minor tweaks from generation to generation, since externally a CMP looks essentially the same from generation to generation, even as the number of cores within it increases. The only real difference is that the board will need to deal with higher memory and I/O bandwidth requirements as the CMPs scale, and slowly change to accommodate new I/O standards as they appear. Over several silicon process generations, the savings in engineering costs can be very significant, because it is relatively easy to simply stamp down a few more cores each time. Also, the same engineering effort can be amortized across a large family of related processors. Simply varying the numbers and clock frequencies of processors can allow essentially the same hardware to function at many different price and performance points.

Of course, since the separate processors on a CMP are visible to programmers as separate entities, we have replaced the old model for programmers with a new parallel programming model. With this kind of model, programmers must divide up their applications into semi-independent parts, or threads, that can operate simultaneously across the processors within a system, or their programs will not be able to take advantage of the processing power inherent in the CMP’s design. Once threading has been performed, programs can take advantage of thread-level parallelism (TLP) by running the separate threads in parallel, in addition to exploiting ILP among individual instructions within each thread. Unfortunately, different types of applications written to target “conventional” Von Neumann uniprocessors respond to these efforts with varying degrees of success.

1.2. THE APPLICATION PARALLELISM LANDSCAPE

To better understand the potential of CMPs, we survey the parallelism in applications. Figure 1.7 shows a graph of the landscape of parallelism that exists in some typical applications. The X-axis shows the various conceptual levels of program parallelism, while the Y -axis shows the granularity of parallelism, which is the average size of each parallel block of machine instructions between communication and/or synchronization points. The graph shows that as the conceptual level of parallelism rises, the granularity of parallelism also tends to increase although there is a significant overlap in granularity between the different levels.[2]

- **Instruction:** All applications possess some parallelism among individual instructions in the application. This level is not illustrated in the figure, since its granularity is simply single instructions. As was discussed previously, superscalar architectures can take advantage of this type of parallelism.

- **Basic Block:** Small groups of instructions terminated by a branch are known as basic blocks. Traditional architectures have not been able to exploit these usefully to extract any parallelism other than by using ILP extraction among instructions within these small blocks. Effective branch prediction has allowed ILP extraction to be applied across a few basic blocks at once, however, greatly increasing the potential for superscalar architectures to find potentially parallel instructions from several basic blocks simultaneously.

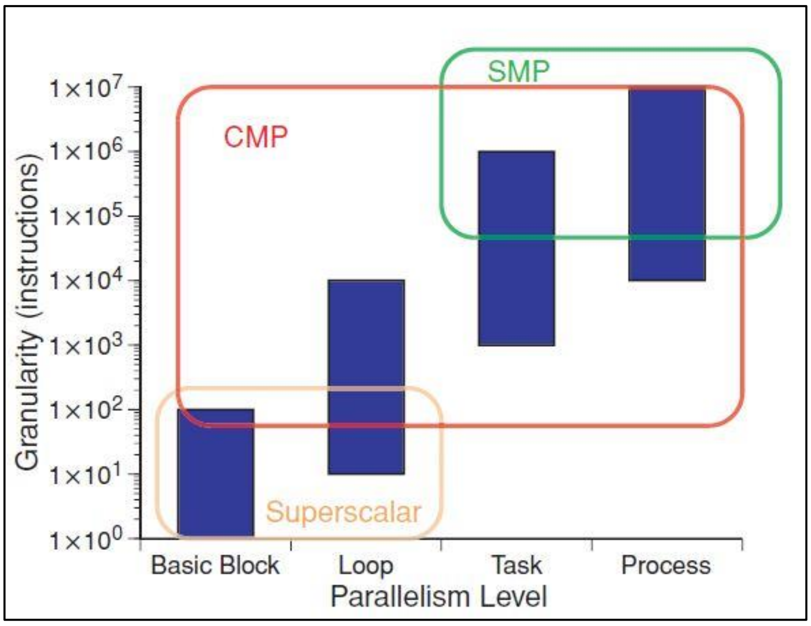


Figure 1.7. A summary of the various "ranges" of parallelism that different processor architectures may attempt to exploit [2]

- **Loop Iterations:** Each iteration of a typical loop often works with independent data elements, and is therefore an independent chunk of parallel work. (This obviously does not apply to loops with highly dependent loop iterations, such as ones doing pointer-chasing.) On conventional systems, the only way to take advantage of this kind of parallelism is to have a superscalar processor with an instruction window large enough to find parallelism among the individual instructions in multiple loop iterations simultaneously, or a compiler smart enough to interleave instructions from different loop iterations together through an optimization known as software pipelining, since hardware cannot parallelize loops directly. Using software tools such as OpenMP, programmers have only had limited success extracting TLP at this level because the loops must be extremely parallel to be divisible into sufficiently large chunks of independent code.

- **Tasks:** Large, independent functions extracted from a single application are known as tasks. For example, word processors today often have background tasks to perform spell checking as you type, and web servers typically allocate each page request coming in from the network to its own task. Unlike the previous types of parallelism, only large-scale symmetric multiprocessor (SMP) architectures composed of multiple microprocessor chips have really been able to exploit this level of parallelism, by having programmers manually divide their code into threads that can explicitly exploit TLP using software mechanisms such as POSIX threads (pthreads), since the parallelism is at far too large a scale for superscalar processors to exploit at the ILP level.

- **Processes:** Beyond tasks are completely independent OS processes, all from different applications and each with their own separate virtual address space. Exploiting parallelism at this level is much like exploiting parallelism among tasks, except that the granularity is even larger.

The measure of application performance at the basic block and loop level is usually defined in terms of the latency of each task, while at the higher task or process levels performance is usually measured using the throughput across multiple tasks or applications, since usually programmers are more interested in the number of tasks completed per unit time than the amount of time allotted to each task.

The advent of CMPs changes the application parallelism landscape. Unlike conventional uniprocessors, multicore chips can use TLP, and can therefore also take advantage of threads to utilize parallelism from the traditional large-grain task and process level parallelism province of SMPs. In addition, due to the much lower communication latencies between processor cores and their ability to incorporate new features that take advantage of these short latencies, such as speculative thread mechanisms, CMPs can attack fine-grained parallelism of loops, tasks and even basic blocks.

As chip multiprocessors (CMPs) become increasingly mainstream, architects have likewise become more interested in how best to share a cache hierarchy among multiple simultaneous threads of execution. The complexity of this problem is exacerbated as the number of simultaneous threads grows from two or four to the tens or hundreds. However, there is no consensus in the architectural community on what “best" means in this context. We try to find the best efficient management of shared last level caches in our proposed design and focus on it.

1.3. CACHE ORGANIZATION

1.3.1. CACHE TERMINOLOGY

The cache within the processor is used by the central processing unit of a computer to reduce the average time to access memory. The cache is smaller than memory and the cache is faster than memory. The cache stores copies of the data from the most frequently used main memory locations. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency.

Most modern server CPUs have at least three independent caches:

1. an instruction cache to speed up executable instruction fetch (I-Cache)

2. a data cache to speed up data fetch and store (D-Cache)
3. a translation look aside buffer (TLB) used to speed up virtual-to-physical address translation for both executable instructions and data.

If the particular address is found in the cache, the block of data is sent to the CPU, and the CPU goes about its operation until it requires something else from memory. When the CPU finds what it needs in the cache, **a hit** has occurred. (as shown Figure 1.8.)

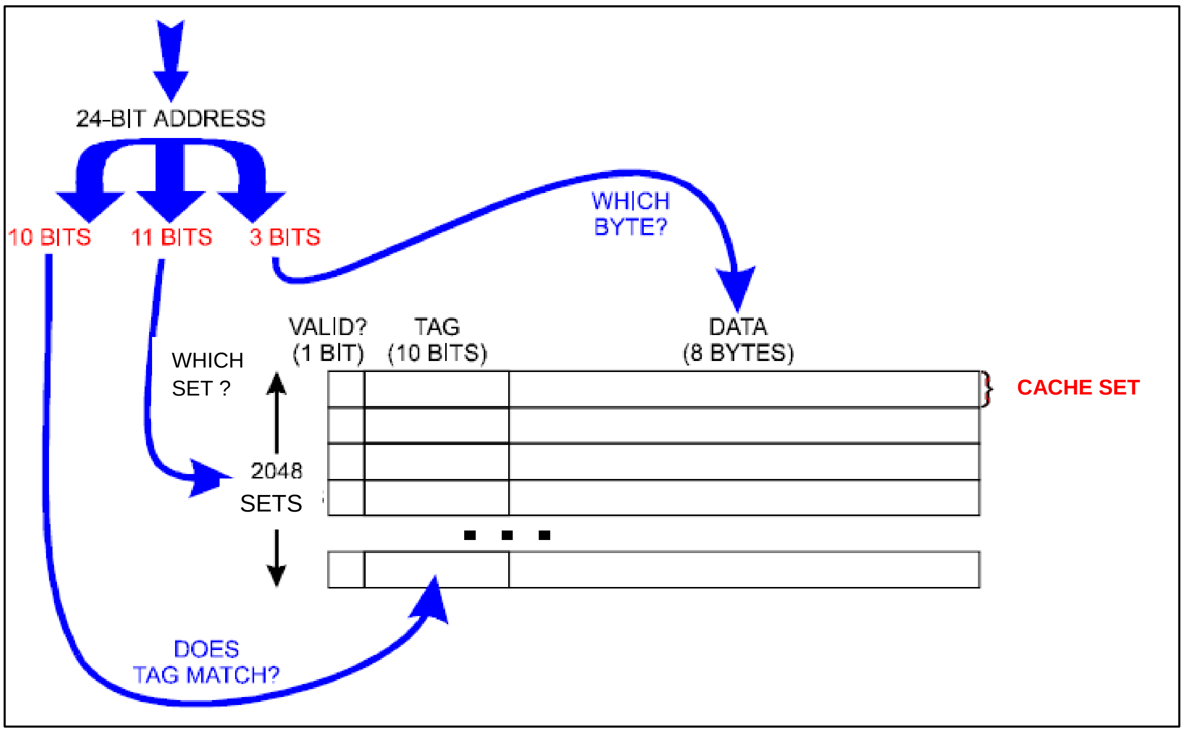


Figure 1.8. Cache hit

When the address requested by the CPU is not in the cache, **a miss** has occurred and the required address along with its block of data is brought into the cache according to how it is mapped. (as shown Figure 1.9.)

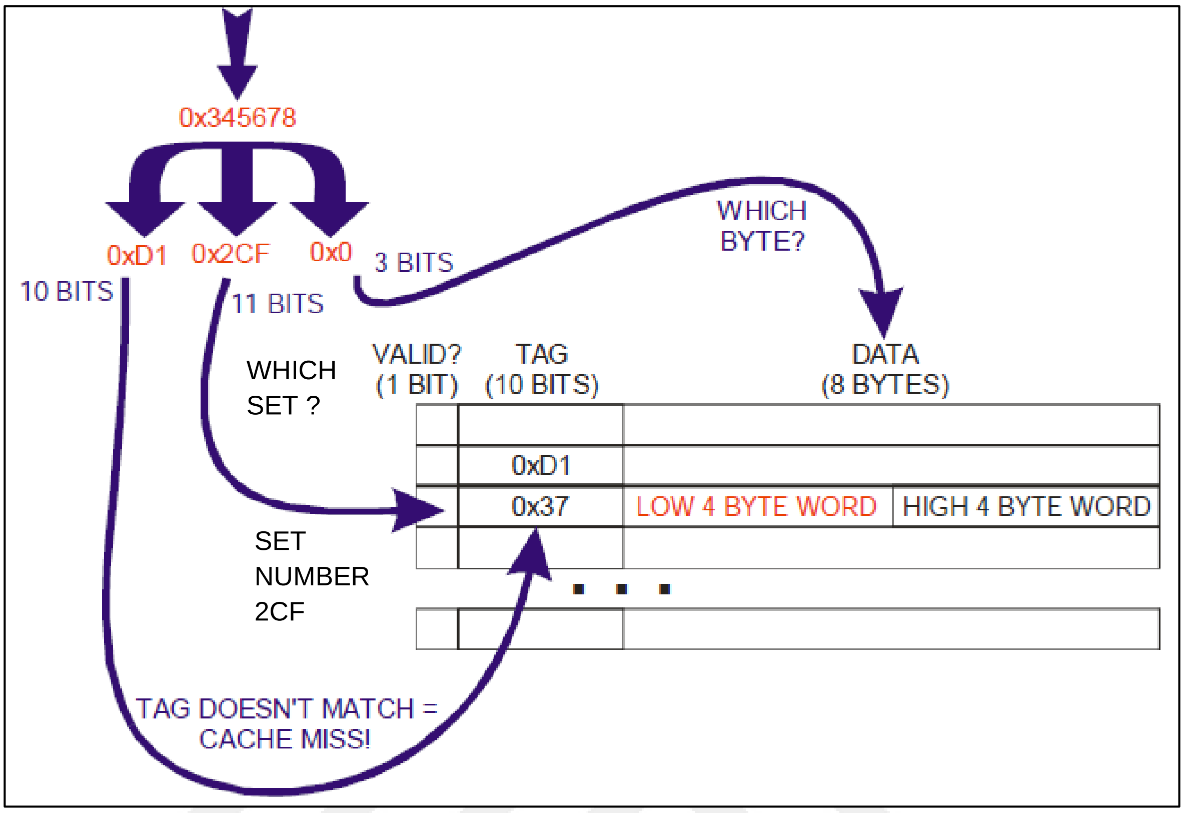


Figure 1.9. Cache miss

1.3.2. CACHE MAPPING TECHNIQUES

Cache mapping is the method by which the contents of main memory are brought into the cache and referenced by the CPU. The mapping method used directly affects the performance of the entire computer system..

Direct Mapping: Main memory locations can only be copied into one location in the cache. This is accomplished by dividing main memory into pages that correspond in size with the cache. Direct mapping is shown in the Figure 1.10.

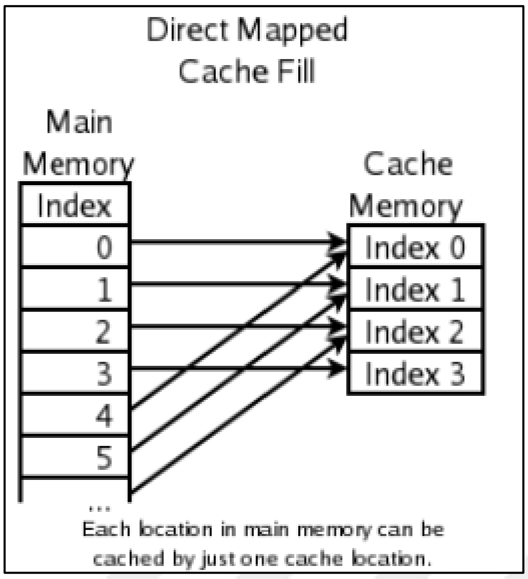


Figure 1.10. Direct Mapping Method

Fully Associative Mapping: Fully associative cache mapping is the most complex, but it is most flexible with regards to where data can reside. A newly read block of main memory can be placed anywhere in a fully associative cache. If the cache is full, a replacement algorithm is used to determine which block in the cache gets replaced by the new data. (Figure 1.11.)

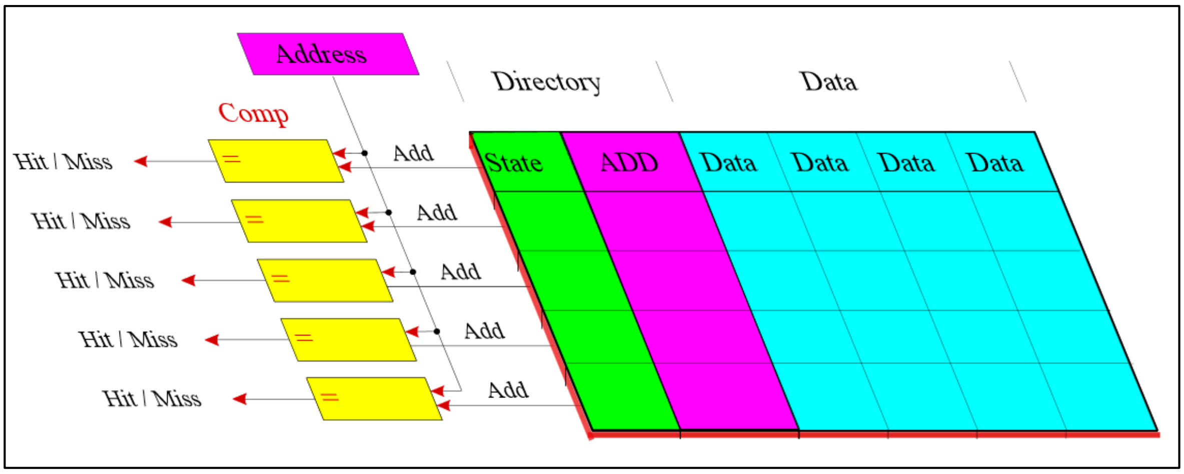


Figure 1.11. Fully associative cache mapping

Set Associative Mapping: Set associative cache mapping combines the best of direct and associative cache mapping techniques. As with a direct mapped cache, blocks of main memory data will still map into as specific set, but they can now be in any N-cache block frames within each set. (See Figure 1.12.)

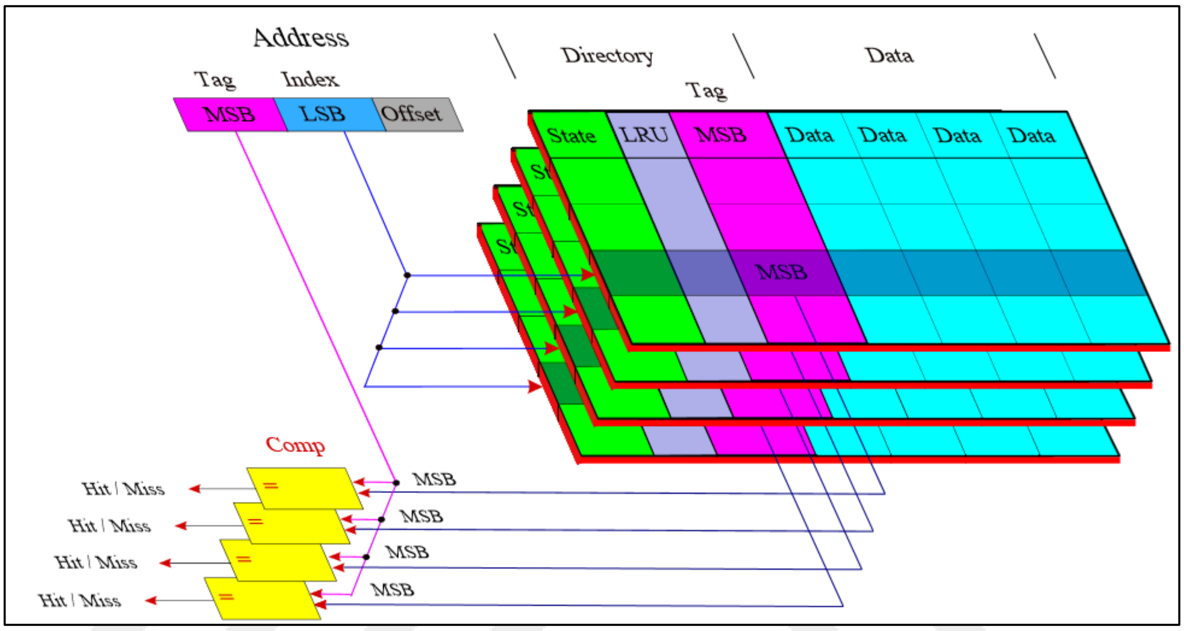


Figure 1.12. Set associative cache mapping

1.3.3. CACHE READ & WRITE, REPLACEMENT

CACHE READ : The two primary methods used to read data from cache and main memory are as follows:

- **Look-through read:** In look-through read, the cache is checked first. If a miss occurs, the reference is sent to main memory to be serviced. This is known as a serial read policy.

- **Look-aside read** A look-aside read presents both cache and main memory with the reference simultaneously. Since the cache will respond faster, if a hit occurs, the request can be terminated before main memory responds. This is known as a parallel read policy.

CACHE REPLACEMENT POLICIES: When new data is read into the cache, a replacement policy determines which block of old data should be replaced. The objective of replacement policies is to retain data that is likely to be used in the near future and discard data that won’t be used immediately. The replacement policies include the following:

- **FIFO:** The first block that was read into cache is the first one to be discarded.
- **LRU:** The block that hasn’t been used in the longest period of time is replaced by the new block.
- **Random:** Blocks are replaced randomly.

CACHE WRITE: Since the cache contents are a duplicate copy of information in main memory, writing (instructions to enter data) to the cache must eventually be made to the same data in main memory. This is done in two ways as follows:

- **Write-through cache:** Writing is made to the corresponding data in both cache and main memory.
- **Write-back cache:** Main memory is not updated until the cache page is returned to main memory.

1.4. MOTIVATION

Nowadays, one of the biggest problems in affecting performance of multi-core processing is memory wall problem or memory bottleneck problem. As is known, each application has its own resource requirements. Moreover, each application may show its own characteristic behavior. For example, despite some applications want too much resource, their performance (IPC) values may be extremely low. Unlike, some applications may show very high performance with very little resources. However, it is not clear that we will be successful when we try to run many applications in different characteristics on the same processor in the last level cache which should be shared by different cores. At this point, we make a preliminary study for answering the following question: when a last level cache is shared in an uncontrolled manner what is its impact on performance? We use the Macsim [14] simulator, we run some representative SPEC 2006 [25] applications which are compiled on a processor at high optimization levels. The details of processor configuration are shown in Table 1.1.

Table 1.1: Multi-core processor configuration

Parametre	**Configuration**
L1 I-Cache (instruction)	64 KB, 64-sets, 128B block, 8-Way, LRU Replacement
L1 D-Cache (data)	64 KB, 128-sets, 64B block, 8- Way, LRU Replacement
L2Cache (instruction+data)	512 KB, 1024-sets, 64B block, 8- Way, LRU Replacement
L3 Cache (shared)	4 MB, 1024-sets, 64B block, 16- Way, 4-tile, 8-way, LRU Replacement

In Figure 1.12, 1.13 and 1.14, we see that the applications that we run in this preliminary study need to access a lot of memory. In the three-dimensional graphs, x-axis represents the sets on the shared last level cache, y-axis represents time periods each of which consists of 655360 (640K) cycles, and z-axis represents number of accesses to the respective sets at certain times. As seen in Figure 1.12, the *gcc* application uses too much of the last level cache and it gains a performance benefit from this. The hit rate of this application to the last level cache is around 75%. In the application presented in figure 1.13, the *lbm* shows completely different characteristics compared to that of *gcc*. This application needs the last level cache much more than *gcc*. However, when this application access to an address, it does not reach to the same address and, therefore, it cannot benefit from the cache but pollute it. In that case, if these two applications work together in an uncontrolled manner, *gcc* will always be a loser. *Lbm* will not lose any performance in case of a pollution of the cache. Because, *lbm* appears to be using the cache but actually it is not. *Gcc* application, on the other hand, experiences huge miss rates and lower performance results because *lbm* contaminates the entire cache. In Figure 1.14, we see the cache activity when *gcc* and *lbm* work together. Table 1.2 shows the performance impact of this mutual run on each of these benchmarks.

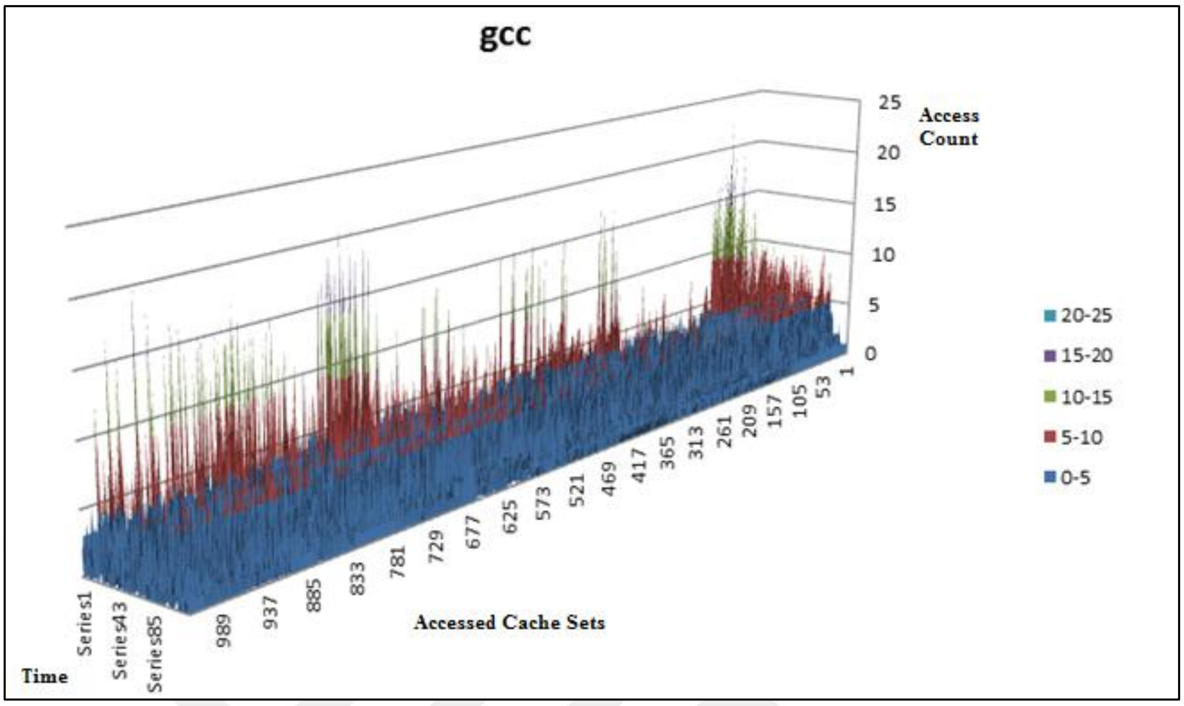


Figure 1.13. Cache Statistics for “*gcc*”

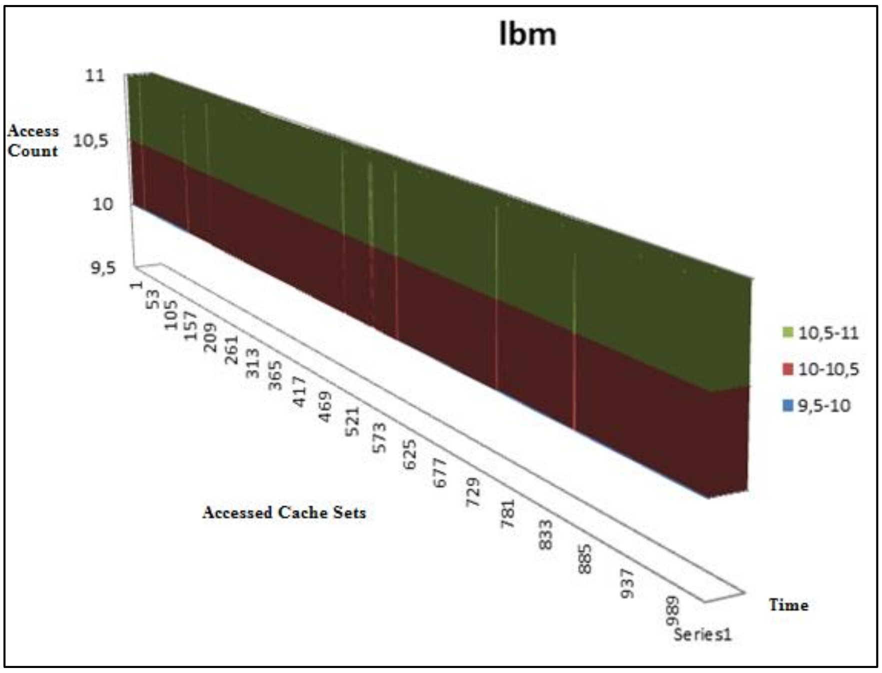


Figure 1.14. Cache Statistics for “*lbm*”

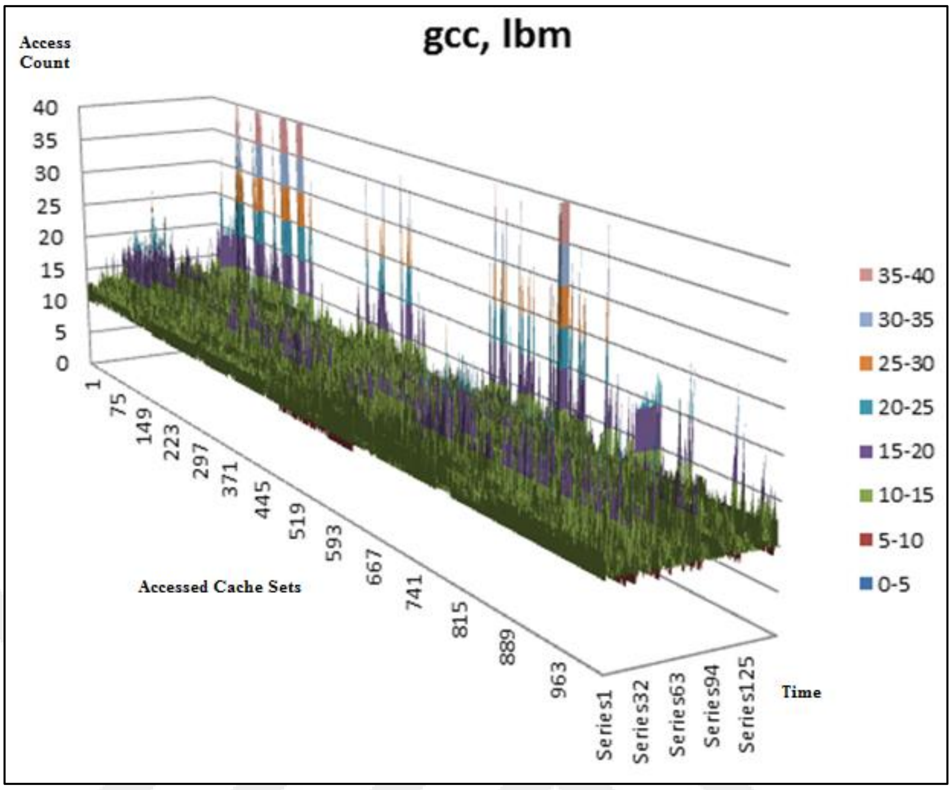


Figure 1.15. Cache Statistics for "*lbm*" and "*gcc*" Mixture

Table 1.2. Performance loss for Mixture "*gcc*" and "*lbm*"

Application	**Performance Loss**
"gcc"	9.8%
"lbm"	%0

When the above scenario is considered, one can think that punishing the harmful *lbm* application, which has 0% hit rate to the last level cache, might be a good idea. Actually, this may be an effective solution for such a case, but there are also counter-examples that exist. In Figure 1.15, *mcf* replaces *lbm*. As it can be seen from this graph, *mcf* application also pollutes the entire cache. The hit rate of the *mcf* is about 8%. However, when we run one *gcc* application and 3 *mcf* applications together, none of the applications receive any performance penalty, as the details are shown in Table 1.3. If we prevent the *mcf* application accessing to the last level cache, performance degradation of this application is less than 1%. So in that case, unlike *lbm*, *mcf* should be classified as a harmless application, and we need a mechanism that does not punish such applications.

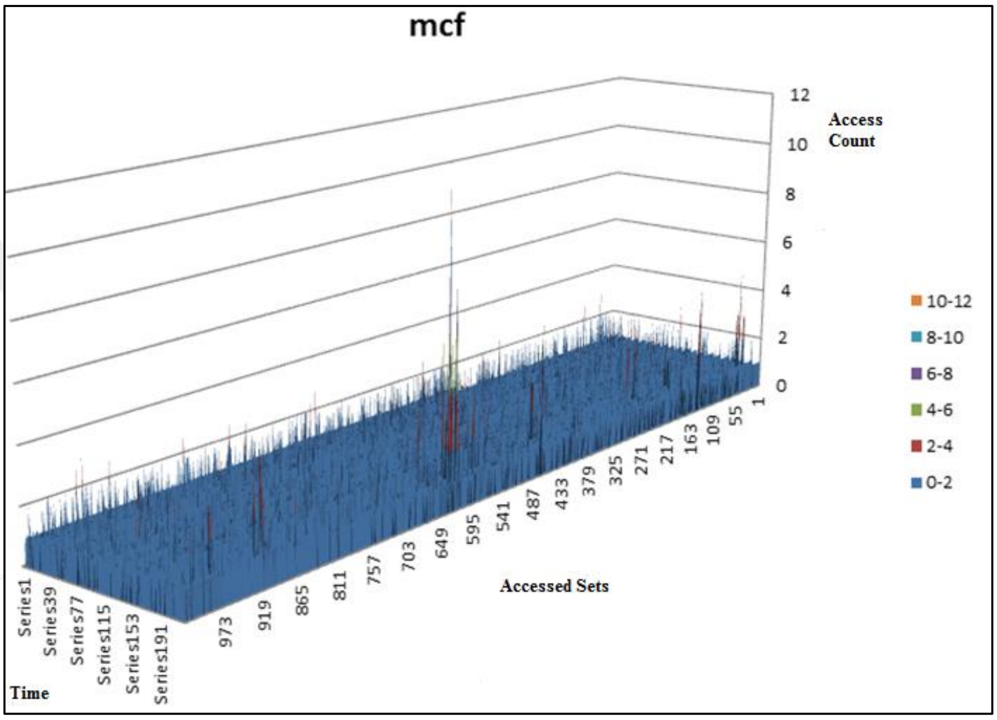


Figure 1.16 Cache Statistics for “mcf”

Table 1.3 Performance loss for Mixture “gcc” and three “mcf”s

Application	Performance Loss
“gcc”	0%
“mcf #1”	0%
“mcf #2”	0%
“mcf #3”	0%

2. LITERATURE SURVEY

A simple solution, which prevents starvation of threads while sharing of the last level cache, is known as **the static partitioning**. In this approach, a portion of the shared cache is dedicated to a core. The size and the location of each partition are fixed and are not changed at run time. Unfortunately, assigning fixed partitions to applications and hoping for the best is not a good idea; a memory-intensive application always needs more cache space whereas a computational-intensive application may not initiate any cache access at all. In such cases inefficient resource usage is imminent.

The dynamic partitioning of shared caches is surely more efficient technique than the static partitioning. In literature, there are several significant research attention. Multiple cores can be allowed to share a cache by allocating each core a portion of the cache space. This partitioning can be done either at the coarser granularity of **cache ways (**this method known as **Way-Based Cache Partitioning)**, as done in [6, 9, 10, 14], or at the finer granularity of **cache sets or blocks (**this method known as **Set-Based Cache Partitioning)**, as done in Vantage [13].

Way-partitioning is popular because of its simplicity of design. It allows for different performance goals like hit maximization [10] and fair sharing [6] to be enforced without introducing much additional hardware complexity. However way-partitioning can be inefficient as it only allows partition sizes to grow or shrink by a fixed large size (inversely proportional to the associativity) while it is possible that the optimal size for a partition falls in between. As the number of cores increases and becomes comparable to the number of ways, such scenarios are likely to occur more frequently. So, this method has serious drawbacks in terms of performance and scalability. Especially, when it is assumed that there will be many-core processors, such as Intel MIC, in the near future, the most recently proposed studies are moving away from this method and searching more scalable mechanisms. Here are the disadvantages of way-based partitioning briefly:

- Limited to coarse-grain allocations
- Only support few partitions
- Reduce cache associativity

- No scalibility
- Low performance

There are many advantages of a set-based cache-partitioning mechanism compared to a way-based partitioning mechanism. We also use this technique in our proposed design. Here, we list some of them which motivate our study:

- Finer-grain control on a typical last level cache: There are much more cache sets than cache ways. When the caches are partitioned on ways, the minimum resizing amount is set in a much coarser-grain. If the application requires only a part of this additional resource, oscillations may be observed in the control mechanism when resource downsizing and upsizing decisions are taken.

- Cache policy freedom and keeping cache structure as it is: When the cache ways are assigned to different applications, the default cache policies and organization can no longer be used. On the contrast, when a set-based partitioning is utilized, no modifications are needed on an existing cache organization.

- Minimum additional circuitry: In a way-based partitioning scheme, each cache way requires multiple counters and wires to collect way-based statistics. That means there is a limit for the number of ways each way-based partitioning mechanism can ideally support. In a set-based scheme, even a fully-associative cache configuration might be feasible, and the number of counters necessary to collect statistics is limited only by the number of cores.

In literature, Utility-Based Cache Partitioning (UCP) [10] uses way-based cache partitioning. This figure shows the framework to support UCP between two applications that execute together on a dual-core system. One of the two applications execute on CORE1 and the other on CORE2. Each core is assigned a utility monitoring (UMON) circuit that tracks the utility information of the application executing on it. The UMON circuit is separated from the shared cache, which allows the UMON circuit to obtain utility information about an application for all the ways in the cache, independent of the contention from the application executing on the other core. The partitioning algorithm uses the information collected by the UMON to decide the number of ways to allocate to each core. The replacement engine of the shared cache is augmented to support the partitions allocated by the partitioning algorithm. (See Figure 2.1.)

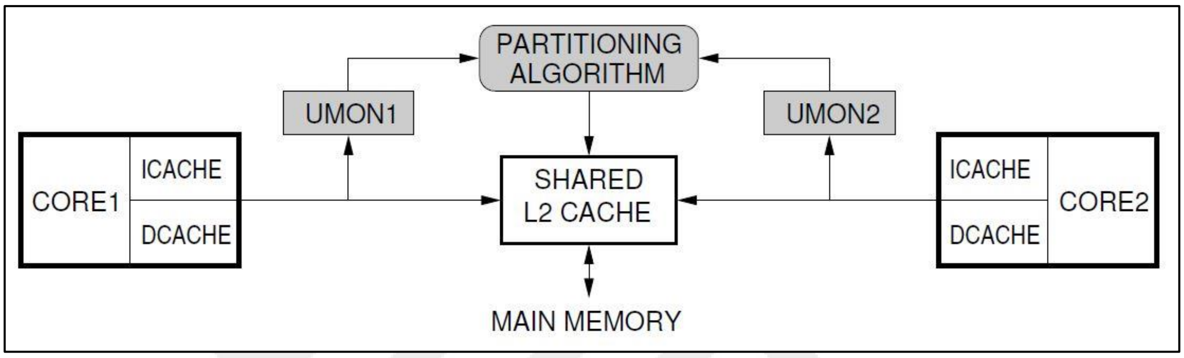


Figure 2.1. Hardware implementation of Utility-Based Cache Partitioning [10]

The partitioning algorithm reads the hit counters from all the UMON circuits of each of the competing applications. The partitioning algorithm tries to minimize the total number of misses incurred by all the applications. The utility information in the hit counters directly correlates with the reduction in misses for a given application when given a fixed number of ways. Thus, reducing the most number of misses is equivalent to maximizing the combined utility. If A and B are two applications with utility functions UA and UB respectively, then for partitioning decisions, the combined utility (Utot) of A and B is computed for all possible partitions for the baseline 16-way cache:

$$U_{tot}(a) = UA_1^i + UB_1^{16-i} \ldots \text{ FOR i =1 to (16-1)} \qquad (2.1)$$

The partition that gives the maximum value for Utot is selected. In our studies, we guarantee that the partitioning algorithm gives at least one way to each application. We invoke the partitioning algorithm once every five million cycles (a design choice based on simulation results). After each partitioning interval, the hit counters in all UMONs are halved. This allows the UMON to retain past information while giving importance to recent information.

The dynamic partitioning of shared caches is first investigated by Suh et al [4]. The proposed study is based on a low overhead, online memory monitoring scheme utilizing a set of hardware counters. The counters indicate the marginal gain in cache hits as the size of the cache is increased. The study suggests a partition module, which uses a greedy algorithm to allocate each cache block to a process that obtains maximum marginal gain by having one additional block.

Stone et al. [18] developed a model for studying the optimal allocation of cache memory among multiple access streams. They experimentally determine a miss rate curve that maps cache size to miss rate for a reference stream, and then fit that curve to an exponential function. Noting that the total miss rate for a pair of memory access streams is the average of the two contributing miss rates, they point out that solving for a minimal miss rate merely involves taking the derivative of this equation, setting it to zero, and solving. They also showed that LRU typically comes close to achieving optimal performance. They focused on partitioning a cache between the instruction and data access streams of a single workload, and did not consider partitioning across multiple workloads.

Thiebaut et al [23] build on Stone's work to partition disk caches for maximal hit ratios. They utilize shadow tags, which are tags without data, to indicate hits that could have occurred had there been a larger allocation. Using this information, they calculate the marginal gain of adjusting the cache allocation. They note that a problem with implementing
a greedy marginal gain algorithm with this methodology is actually finding the memory stream with the largest marginal gain, since the functions are non-monotonic. They resort to performing a sort every time they update a partition. Their study assumed fully associative disk caches and partitioning on a disk block granularity. This amount of computation is likely too hefty for a CMP, while it is acceptable for a long latency entity like a disk cache.

Hsu et al. [5] examine various cache policies such as communist and utilitarian policies. The communist policy tends to achieve fairness rather than maximizing the performance for running threads. The utilitarian policy tends to do just the opposite. The authors propose the usage of instruction per cycle (IPC), misses per access matrices and weighted IPC metric, and conclude that using a traditional cache replacement policy such as LRU and performing static cache partitioning is not sufficient to provide near optimal performance. They state that a thread-aware cache resource allocation mechanism is required, and the sole use of communist or utilitarian policy for partitioning cache in CMP may not work perfectly for some type of workloads.

Chiou et al. [19] propose a partitioning scheme that partitions at the granularity of cache ways. Partitioning is achieved by limiting the cache ways in which a thread can place its data. The exibility in placement is thus limited.

Settle et al. [7] also investigate dynamic cache partitioning mechanism based on cache ways. The partition control mechanism gives large percentage of available cache storage to applications with high degree of global data reuse to increase chances of process utilization. When the thread id of a cache request differs from that of the normal LRU candidate, the cache controller checks the reuse of the candidate line to determine its potential for harming the system performance. The reuse is simply the cache access frequency counter used in least frequently used (LFU) cache replacement policies. If the reuse rank of a line is higher than a threshold value, the line is not considered for eviction. This algorithm increases the time that data from another thread stays in the cache. Thus, in case where one thread has a very high cache access frequency, this technique will make it less likely for the high frequency thread to evict important data belonging to another thread that accesses the cache much less often.

Kim et al. [6] present a cache partitioning algorithm which focuses on fairness in a small scale CMP using SPEC2000 benchmarks. They evaluate several metrics and correlate them to execution time to determine what a good online metric to drive their policy decisions should be, and develop an algorithm that attempts to keep those metrics as equal as possible throughout execution time.

Lin et al. [8] propose partitioning the cache based on an O/S technique called page coloring. A page color consists of several common bits between the cache index and the physical page number in the physical address. A physically addressed cache is divided into non-intersecting regions by page color, and pages with the same color are mapped to the same cache region. By assigning different page colors to different processes, the cache space is partitioned between cores.

Chandra et al. [24] deal not with partitioning but with predicting inter-thread cache contention in a shared cache on a CMP, with the intent to use this information to prevent thrashing in a shared cache. They present a mechanism to accurately predict when threads will thrash, though they do not present a means to prevent it.

Rafique et al. [9] propose using a hardware quota enforcement mechanism tomanage shared caches in CMP while a communication between the hardware and O/S establish to apply a variety of policies by tuning the quotas during regularly scheduled O/S interventions. Disadvantage of this work is the limitations of the proposed hardware mechanism that only supports a coarse granularity of cache allocation.

Qureshi and Patt [10] partition the cache-ways dynamically among competing applications. They propose a low overhead utility hardware circuit that monitors the reduction in misses for each application for a given amount of cache resource. Later, they collect the information by a circuit named utility monitor (UMON) used for deciding the amount of cache resources that each application need for periodic intervals. Using the monitored statistics, UMON can derive the optimal L2 cache partition that would minimize the total number of misses, as this number corresponds to the sum of misses of each thread with the assigned number of ways.

Moreto et al. [20] propose a dynamic cache partitioning mechanism to maximize the total throughput of running threads by minimizing the total cost. The algorithm assigns higher costs to isolated L2 misses due to their higher impact on performance, and assigns lower costs to clustered L2 misses. The cost assigning process is implemented by extra hardware, which are auxiliary tag directory (ATD), miss status holding register (MSHR) and hit status holding register (HSHR). The job of ADT is to keep track of the L2 accesses for any possible cache configuration. MSHR and HSHR are used to compute the memory level parallelism (MLP) cost of the access. Moreover, they also keep track of stack distance of each access. Based on the gathered information from the MLP cost and stack distance, a performance benefit of converting L2 misses into hits when assigning more ways to a thread is estimated.

Sanchez and Kozyrakis [13] introduce a cache partitioning scheme named Vantage, which maintains high associativity and strong isolation among partitions. Their mechanism maintains the size of each partition by matching the average rates at which lines enter and leave a partition. The authors claim that Vantage works best with a special cache architecture, Zcache [12]. However, they also indicate that the proposed mechanism may work with a 16-way set associative cache with less promising results.

Iyer [21] presents a framework for providing differentiated services to various threads via the cache hierarchy in a CMP. The framework consists of classifying heterogeneous memory streams, assigning priorities, and enforcing them. He presents several means of enforcing a partition, including selective allocation and set partitioning.

Fedorova et al. [26] present an operating system scheduling algorithm to deduce which sets of threads would coexist the best to schedule at the same time. Their goal is to schedule threads which would yield the lowest overall miss rates without starving any threads. This technique may not be relevant on large scale CMPs where the number of software threads may not outnumber the number of hardware threads by so much as to make scheduling an issue. Furthermore, large scale CMPs will likely support multiple virtual machines, making system-wide optimization outside the scope of any one OS scheduler.

One of the few studies that focus on set-based scalable partitioning of cache resources is Vantage, and it does cache block replacement by utilizing multi-level complex hash functions and arranges partition sizes by regulating the number of addresses inserted into the cache and number of addresses that are evicted from the cache. Besides, this method can work well with a cache known as Zcache [12]. The team also reports that they tried to integrate their proposed design with a standard cache and they cannot come close to the results they had with the Zcache.

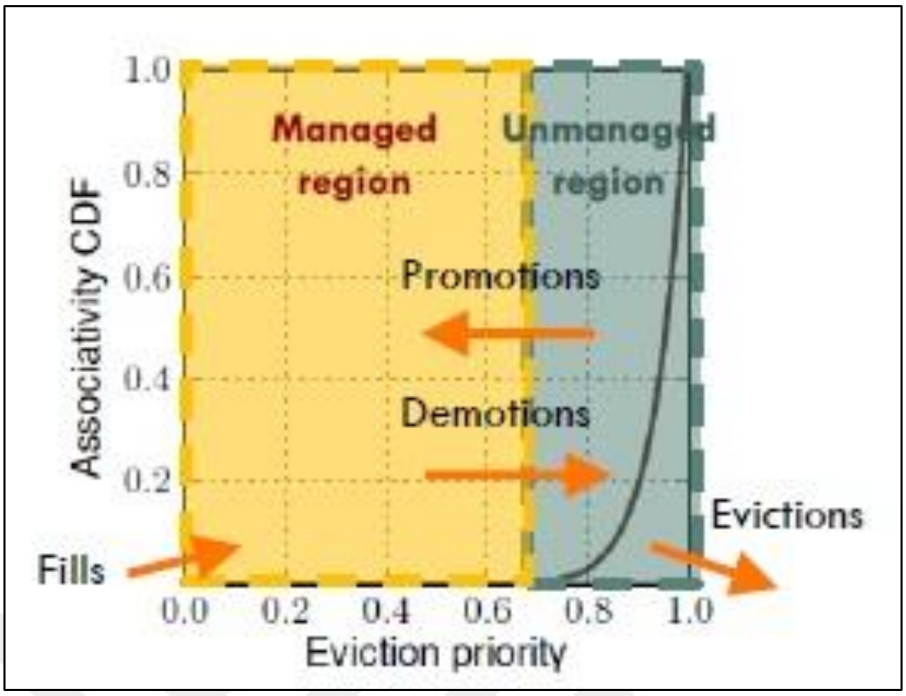


Figure 2.2. Vantage managed-unmanaged region division [12]

Vantage [12] is the only known scheme that proposes a fine grained partitioning framework. However it achieves this only for a portion and not all of the cache and does so with significant changes to the hardware. Vantage on the other hand logically partitions the cache into ‘managed’ and ‘unmanaged’ regions and achieves the desired target occupancy in the ‘managed’ portion of the cache by borrowing space from the unmanaged region. This is a fundamental change to the cache organization. Also vantage requires the replacement policies, including commonly used LRU, to be implemented in a Vantage-friendly fashion

Manikantan et al [22] proposes Probabilistic Shared Cache Management (PriSM), a framework to manage the cache occupancy of different cores at cache block granularity by controlling their eviction probabilities. The proposed framework requires only simple hardware changes to implement, can scale to larger core count and is flexible enough to support a variety of performance goals. We demonstrate the flexibility of PriSM by implementing three allocation policies to achieve Hit-Maximization (PriSM-H), Fairness (PriSMF) and QOS (PriSM-Q) in our proposed framework. We demonstrate the scalable nature of our solution by studying its performance from low core count (4-cores) to high core count (32-cores).

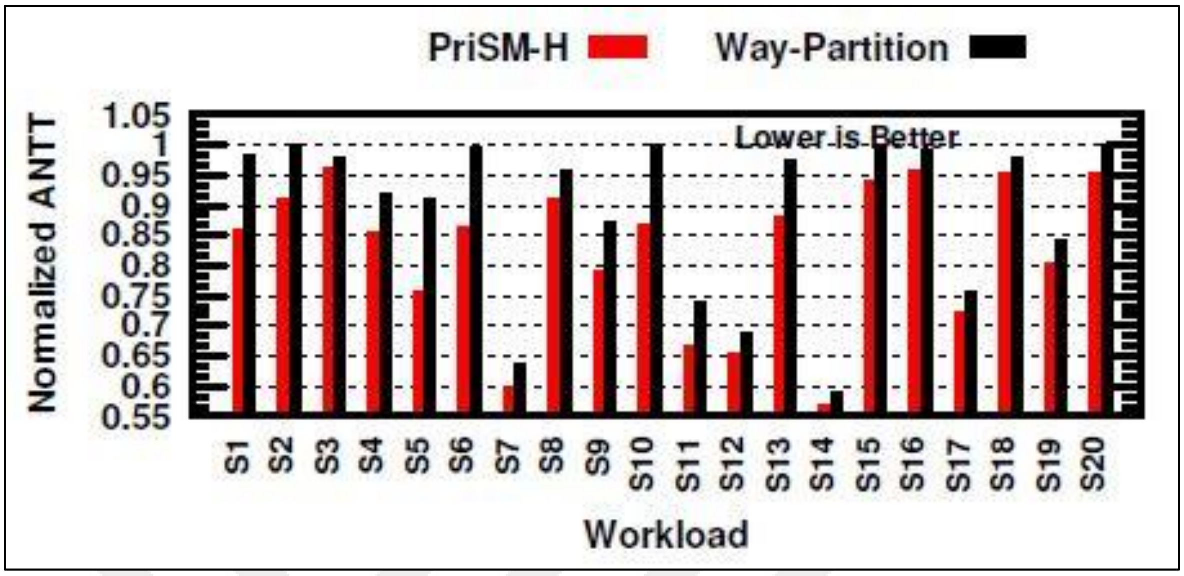


Figure 2.3. HitMaximization: Gains provided by PriSM over waypartitioning [22]

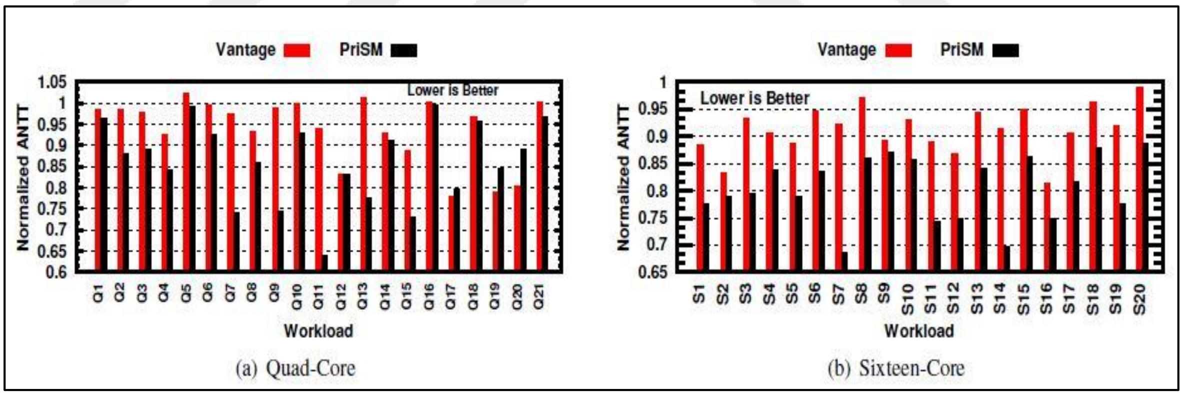


Figure 2.4. Performance of Vantage and PriSM in 4 and 16 core machines [22]

3. SET-BASED DYNAMIC CACHE PARTITIONING ON CHIP MULTIPROCESSORS

This chapter explains the design and implementation details of our proposed design. In briefly, our proposed solution tries to classify applications running on the system according to runtime statistiscs like hit rate, miss rate and cache invalidations. Then, based on the classification of the applications, Partitioner allocates new resources to each application. We have identified 655360 Cycle (640 KCycle) as the threshold value for making three steps below:

On each epoch (640 KCycyle) do the following steps:

Step 1. Until number of cycles reaches a certain threshold, continue to **gather statistics** about the applications running on the system.

Step 2. After this threshold is reached, **classify** the applications according to the data collected as “harmless”, “harmful” or “very harmful” or “no operation”.

Step 3. Based on the result of the classification process, **Partitioner** decides how to distribute the cache among the applications.

In Figure 3.1, a feedback mechanism that is needed to realize the above mentioned steps is shown. As can be seen from the figure, our proposed design is settled in front of the last level cache to provide access control with the help of a structure, which we call the Partition Map. This circuit can work with all kinds of last level cache organizations.

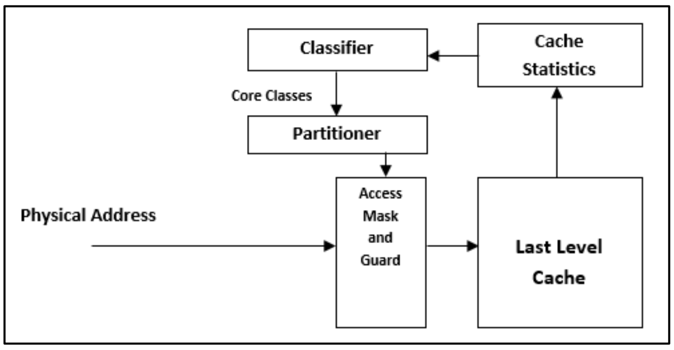


Figure 3.1. Our proposed design cache organization

3.1. CACHE STATISTICS

In order to collect cache statistics, we partition the last level cache as follows:

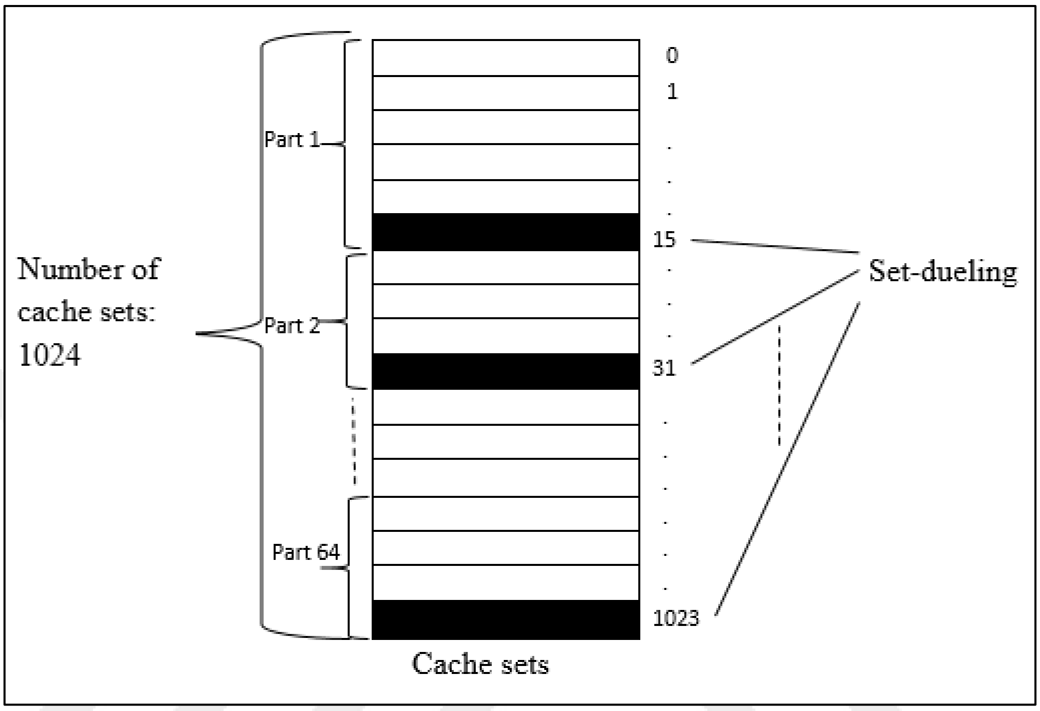


Figure 3.2. Logic cache partitioning of our design

In the figure 3.2, we select the last set of each part for set-dueling on LLC (shown in black). We allow all cores to access these sets. At any time, any core can access these duel sets. We calculate Steal Rate, Miss Rate and Traffic value using these duel sets.

Traffic: We calculate this value for an each core. This parameter reports the number of access per core on duel sets.

$$Traffic = \#\, of\ access\ for\ that\ core \tag{3.1}$$

Steal Rate (%) : This parameter indicates how much an application steal cache blocks from other applications on duel sets. This value shows us, the ratio of the number of cache sets stolen by the application to total traffic of duel sets for that application. We calculate this value using the equation shown in below:

$$Steal\ Rate\ (\%) = \frac{\#\ of\ steal\ from\ a\ core}{\#\ of\ access\ for\ that\ core} = \frac{\#\ of\ steal\ from\ a\ core}{Traffic} \qquad (3.2)$$

Miss Rate (%) : If the data is not in cache, a miss has occurred and then we bring the data from memory to cache. This parameter reports the ratio of the number of cache sets missed by the application to total traffic of duel sets for that application.

$$Miss\ Rate\ (\%) = \frac{\#\ of\ miss\ from\ a\ core}{\#\ of\ access\ for\ that\ core} = \frac{\#\ of\ miss\ from\ a\ core}{Traffic} \qquad (3.3)$$

We do not use hit rate, because hit rate is opposite of miss rate. We use only miss rate in our proposed design. We can write the hit rate formula like this:

$$Hit\ Rate\ = 1 - Miss\ Rate \qquad (3.4)$$

In the figure 3.2, the rest of cache sets (shown in white) were assigned to a certain core at any time. At any time, only the owner of these sets can access and use them. If any core which is not an owner want to access the cache set, we don’t allow to access or use it. When this occurs, we increase the value of Attempted Steal Count. At the end, we calculate **Attempted Steal Rate** parameter:

$$Attempted\ Steal\ Rate\ (A) = \frac{\#\ of\ attempted\ steal\ from\ a\ core}{\#\ of\ access\ for\ per\ core\ (not\ only\ duel\ set\ traffic)} \qquad (3.5)$$

This parameter indicates how much an application attempts to steal cache blocks from other applications. In our proposed design, we prevent applications to access partitions that do not belong to them. This value, therefore, is not correlated to the previous parameter, steal rate.

At the end, all of these parameters are sent to our Classifier circuitry as inputs. Cache control mechanism is required to view these parameters periodically to come up with accurate partitioning decisions. At the end of each period, the Classifier is triggered and the cache is allocated by the Partitioner, accordingly.

3.2. CLASSIFIER ALGORITHM

Our classifier algorithm takes as inputs these values: Attempted Steal Rate (%), Steal Rate (%), Miss Rate (%) and Traffic value. Consequently, it classifies them according to the algorithm shown below:

If the attempted steal rate of an application is higher than 30%, we set the value of parameter A to 1. In other cases we set it to 0. Here is the program flow in order to transform of Attempted Steal Rate to Attempted Steal (A):

A → Attempted Steal

```
if Attempted Steal Rate (%) > 0.30 then
    A = 1;
else
    A = 0;
end
```

S is the steal rate. If the steal rate of an application is higher than 75%, we set the value of parameter S to 2. If the steal rate of an application is between 50% and 75%, we set the value of parameter S to 1. In other cases we set it to 0.

S → Steal

```
if Steal Rate (%) > 0.75 then
    S = 2;
else if Steal Rate (%) > 0.50 then
    S = 1;
else
    S = 0;
end
```

If the miss rate of an application is higher than 75%, we set the value of parameter M to 2. If the miss rate of an application is between 50% and 75%, we set the value of parameter M to 1. In other cases we set it to 0. Again, these are the thresholds obtained from our preliminary empirical study.

M → Miss

```
if Miss Rate (%) > 0.75 then
    M = 2;
else if Miss Rate (%) > 0.50 then
    M = 1;
else
    M = 0;
end
```

T parameter indicates the degree of traffic in sets duel. If any application access over 100 to set duel, we set the value of parameter T to 2. If an application access between 50 and 100 then we set the value of parameter 1, in other case we set it to 0. Again, these are the thresholds obtained from our preliminary empirical study as described in detail in chapter 1.4.

T → Traffic

```
if Traffic > 100 then
    T = 2;
else if Traffic > 50 then
    T = 1;
else
    T = 0;
end
```

During working hours, characteristics of the applications may be changed, continuously. Likewise, applications which are run by cores may change from time to time. At the end of each epoch, **Classifier Circuit** calculates **total weight** of each application. When we calculate the total weight of an application, we use this equation shown below:

$$Weight = M + A + (S * T) + T \qquad (3.6)$$

This formula is obtained from our preliminary empirical study. In this formula, we thought that effect of the Steal (S) value to the core weight should be higher when the value of Traffic (T) is too high. So, we multiply the Steal (S) and the Traffic (T) value with each other.

Then according to this total weight value, we classify the applications into four classes which are harmless, harmful, very harmful and no operation.

This total weight value enables us to classify each application. This value can be between 0 and 9. We decided that if the weight of an application is lower than 2, we called it as *no operation*. If the weight of an application is between 2 and 3, we called it *harmless*. If the weight of an application is between 4 and 6, we called it *harmful*. Otherwise, if the weight of an application is higher than 6 then we called it *very harmful*.

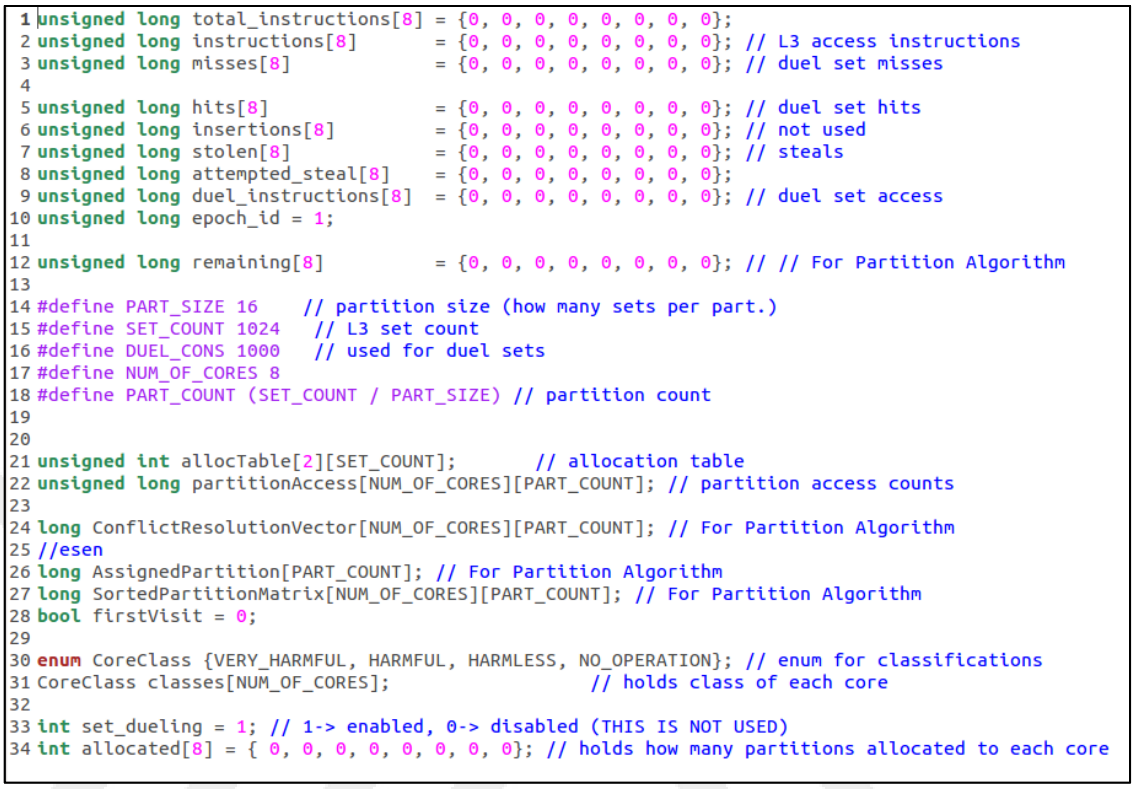
In the future, in order to be able to doing homogeneous distribution of total weight, dynamic distribution will be used. In this approach, if the total weight interval is between 2 and 5, the distribution of above can be different. We can define the value of 5 as very harmful.

Our cache partitioning principle is like that: When an application is classified as "no operation", we don't give any cache sets to this application and we punish it. If an application is "harmful", we give it more cache resources than very harmful one. If an application is "harmless", we give it more cache resources than harmful one.

3.2.1. CLASSIFIER IMPLEMENTATION

First of all, we defined global variables at the beginning of "memory.cc" file in macsim simulator, as you see in the Figure 3.3.

Then, we write a new method which is called "classify" in "memory.cc" in order to implement Classifier. As explained above, firstly, we have to calculate weight of applications before to classify.



```
unsigned long total_instructions[8] = {0, 0, 0, 0, 0, 0, 0, 0};
unsigned long instructions[8]       = {0, 0, 0, 0, 0, 0, 0, 0}; // L3 access instructions
unsigned long misses[8]             = {0, 0, 0, 0, 0, 0, 0, 0}; // duel set misses

unsigned long hits[8]               = {0, 0, 0, 0, 0, 0, 0, 0}; // duel set hits
unsigned long insertions[8]         = {0, 0, 0, 0, 0, 0, 0, 0}; // not used
unsigned long stolen[8]             = {0, 0, 0, 0, 0, 0, 0, 0}; // steals
unsigned long attempted_steal[8]    = {0, 0, 0, 0, 0, 0, 0, 0};
unsigned long duel_instructions[8]  = {0, 0, 0, 0, 0, 0, 0, 0}; // duel set access
unsigned long epoch_id = 1;

unsigned long remaining[8]          = {0, 0, 0, 0, 0, 0, 0, 0}; // // For Partition Algorithm

#define PART_SIZE 16    // partition size (how many sets per part.)
#define SET_COUNT 1024   // L3 set count
#define DUEL_CONS 1000   // used for duel sets
#define NUM_OF_CORES 8
#define PART_COUNT (SET_COUNT / PART_SIZE) // partition count


unsigned int allocTable[2][SET_COUNT];        // allocation table
unsigned long partitionAccess[NUM_OF_CORES][PART_COUNT]; // partition access counts

long ConflictResolutionVector[NUM_OF_CORES][PART_COUNT]; // For Partition Algorithm
//esen
long AssignedPartition[PART_COUNT]; // For Partition Algorithm
long SortedPartitionMatrix[NUM_OF_CORES][PART_COUNT]; // For Partition Algorithm
bool firstVisit = 0;

enum CoreClass {VERY_HARMFUL, HARMFUL, HARMLESS, NO_OPERATION}; // enum for classifications
CoreClass classes[NUM_OF_CORES];                    // holds class of each core

int set_dueling = 1; // 1-> enabled, 0-> disabled (THIS IS NOT USED)
int allocated[8] = { 0, 0, 0, 0, 0, 0, 0, 0}; // holds how many partitions allocated to each core
```

Figure 3.3. Defining Global Variables

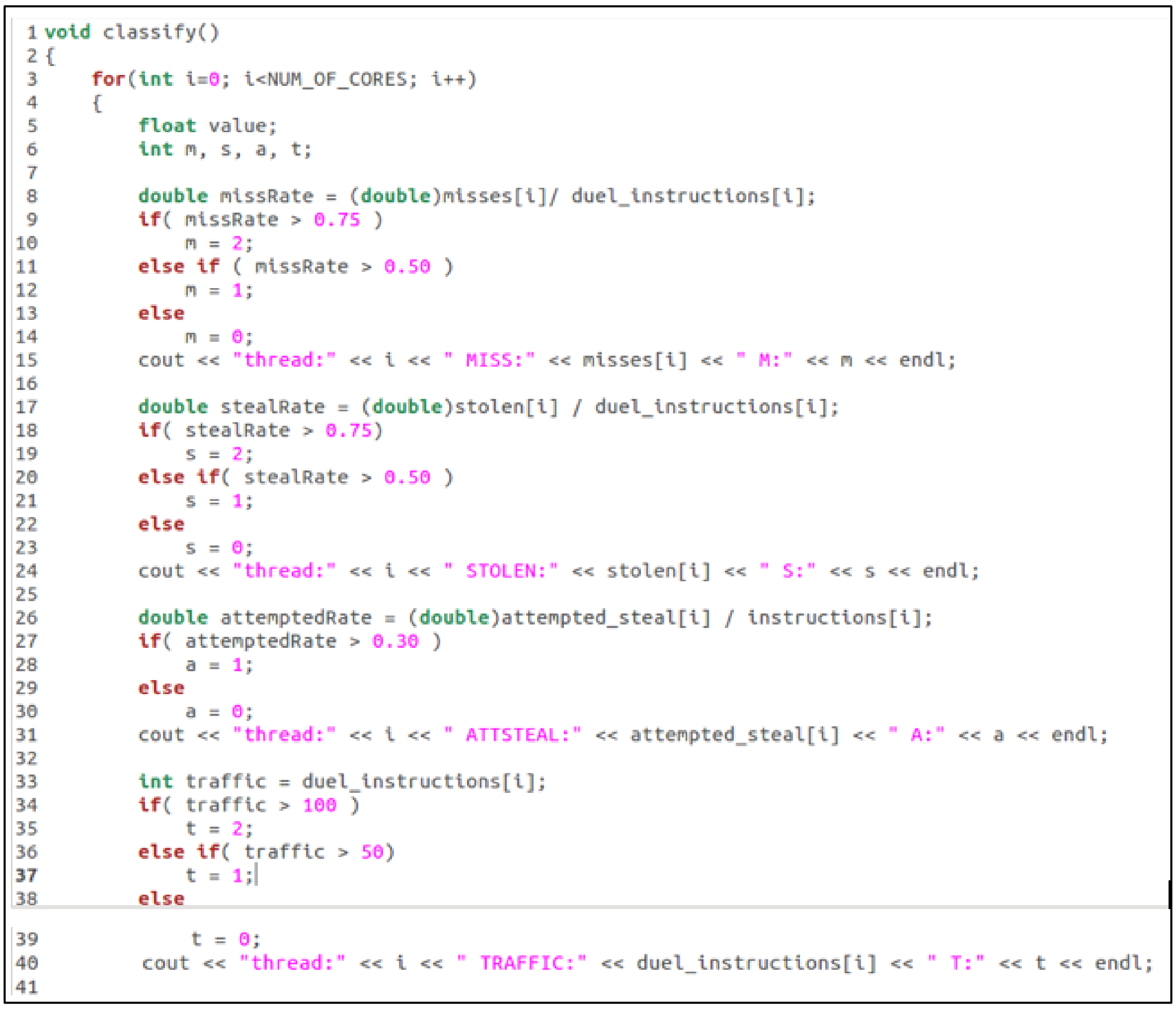
Therefore, firstly we have to calculate miss rate, attempted steal rate, steal rate and traffic.

As seen in Figure 3.4, we use “If” statements to calculate them. If the miss rate of an application is higher than 0.75, we set the value of miss rate to 2. If the miss rate of an application is higher than 0.50, we set the value of miss rate to 1. In other cases we set it to 0.

If the steal rate of an applications is higher than 0.75, we set the value of miss rate to 2. If the steal rate of an application is higher than 0.50, we set the value of steal rate to 1. In other cases we set it to 0.

If the attempted steal rate of an application is higher than 0.30, we set the value of attempted steal rate to 1. In other cases we set it to 0.

If the traffic of an application is higher than 100, we set the value of traffic 2. If the traffic of an application is higher than 50, we set the value of attempted steal rate 1. In other cases we set it to 0.

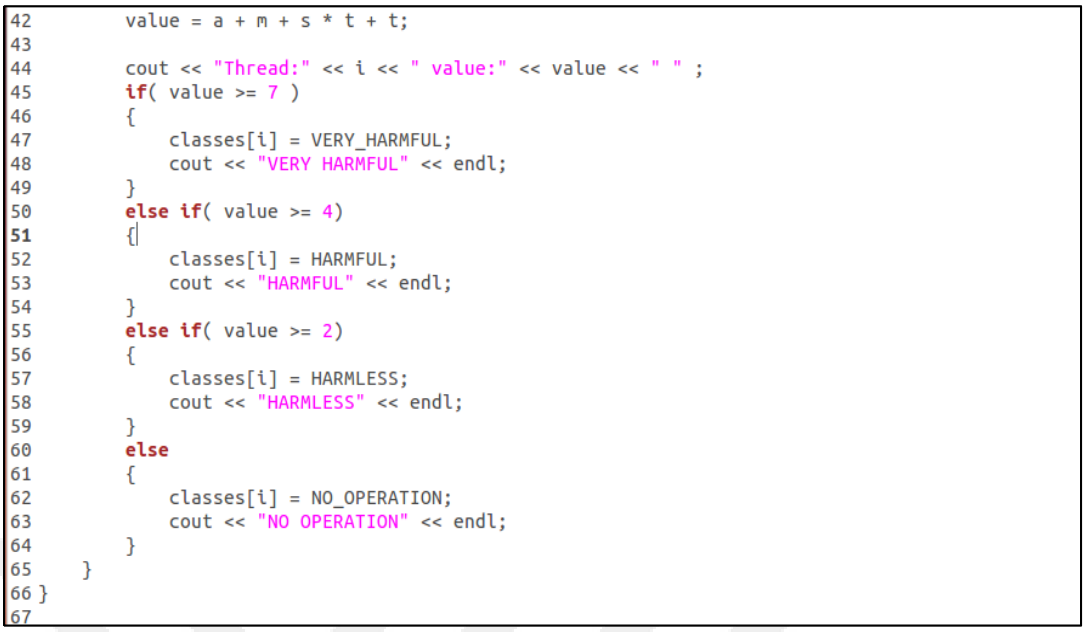


```
void classify()
{
    for(int i=0; i<NUM_OF_CORES; i++)
    {
        float value;
        int m, s, a, t;

        double missRate = (double)misses[i]/ duel_instructions[i];
        if( missRate > 0.75 )
            m = 2;
        else if ( missRate > 0.50 )
            m = 1;
        else
            m = 0;
        cout << "thread:" << i << " MISS:" << misses[i] << " M:" << m << endl;

        double stealRate = (double)stolen[i] / duel_instructions[i];
        if( stealRate > 0.75)
            s = 2;
        else if( stealRate > 0.50 )
            s = 1;
        else
            s = 0;
        cout << "thread:" << i << " STOLEN:" << stolen[i] << " S:" << s << endl;

        double attemptedRate = (double)attempted_steal[i] / instructions[i];
        if( attemptedRate > 0.30 )
            a = 1;
        else
            a = 0;
        cout << "thread:" << i << " ATTSTEAL:" << attempted_steal[i] << " A:" << a << endl;

        int traffic = duel_instructions[i];
        if( traffic > 100 )
            t = 2;
        else if( traffic > 50)
            t = 1;
        else
            t = 0;
        cout << "thread:" << i << " TRAFFIC:" << duel_instructions[i] << " T:" << t << endl;

```

Figure 3.4. Classify() method - 1

As seen in Figure 3.5, firstly we calculate the “value”. Then, again we use “If” statements in order to determine classes of applications.

If the value of application is greater than 6, application can be assigned as “VERY HARMFUL”. If the value of application is greater than 3, application can be assigned as “HARMFUL”. If the value of application is greater than 1, application can be assigned as “HARMLESS”. Other status, application can be assigned as “NO OPERATION”.



```
        value = a + m + s * t + t;

        cout << "Thread:" << i << " value:" << value << " " ;
        if( value >= 7 )
        {
            classes[i] = VERY_HARMFUL;
            cout << "VERY HARMFUL" << endl;
        }
        else if( value >= 4)
        {
            classes[i] = HARMFUL;
            cout << "HARMFUL" << endl;
        }
        else if( value >= 2)
        {
            classes[i] = HARMLESS;
            cout << "HARMLESS" << endl;
        }
        else
        {
            classes[i] = NO_OPERATION;
            cout << "NO OPERATION" << endl;
        }
    }
}
```

Figure 3.5. Classify() method - 2

3.3. PARTITIONER ALGORITHM

The task of the Partitioner is to logically allocate set-based cache partitions among cores by the help of the information coming from the Classifier circuit. As a result, this circuit decides which core is allowed to access which cache partition. At this point, Partitioner must be prepared for all kinds of combinations of core classes and must make the best scheduling effort.

For example, if Classifier reports that four cores are running very harmful applications in a quad-core processor then Partitioner should partition the cache equally among all such applications. If Classifier reports that one core is running a very harmful application and other cores are running harmless applications, then, Partitioner should offer solutions to reduce the damage which may be created by the harmful application.

We created a table for allocating partitions to each application for all combinations of application classes. Table 3.1 provides a sample Partitioner table for a quad-core processor with 64 LLC partitions. In this table, the column named as A0 means that how many LLC partitions we allocate to Core 0. Likewise, A1, A2 and A3 are the same meaning.

Table 3.1. Example of Partitioner Table

Core0	Core1	Core2	Core3	A0	A1	A2	A3
0	0	0	0	16	16	16	16
0	0	0	1	18	17	17	12
0	0	0	2	19	19	18	8
0	0	1	1	20	20	12	12
0	0	1	2	20	20	16	8
0	0	2	2	24	24	8	8
0	1	1	1	28	12	12	12
0	1	1	2	24	16	16	8
0	1	2	2	32	16	8	8
0	2	2	2	40	8	8	8
1	1	1	1	16	16	16	16
1	1	1	2	19	19	18	8
1	1	2	2	24	24	8	8
1	2	2	2	40	8	8	8
2	2	2	2	16	16	16	16

In Table 3.1, “0” indicates that an application is harmless, “1” indicates that an application is harmful, “2” indicates that an application is very harmful and “3” indicates that an application is no operation. As can be seen from the Table, initially we are planning to give minimum number of cache partitions to very harmful applications. Applications that are classified as harmful receive more partitions than the applications that are very harmful but fewer partitions than the applications that are harmless.

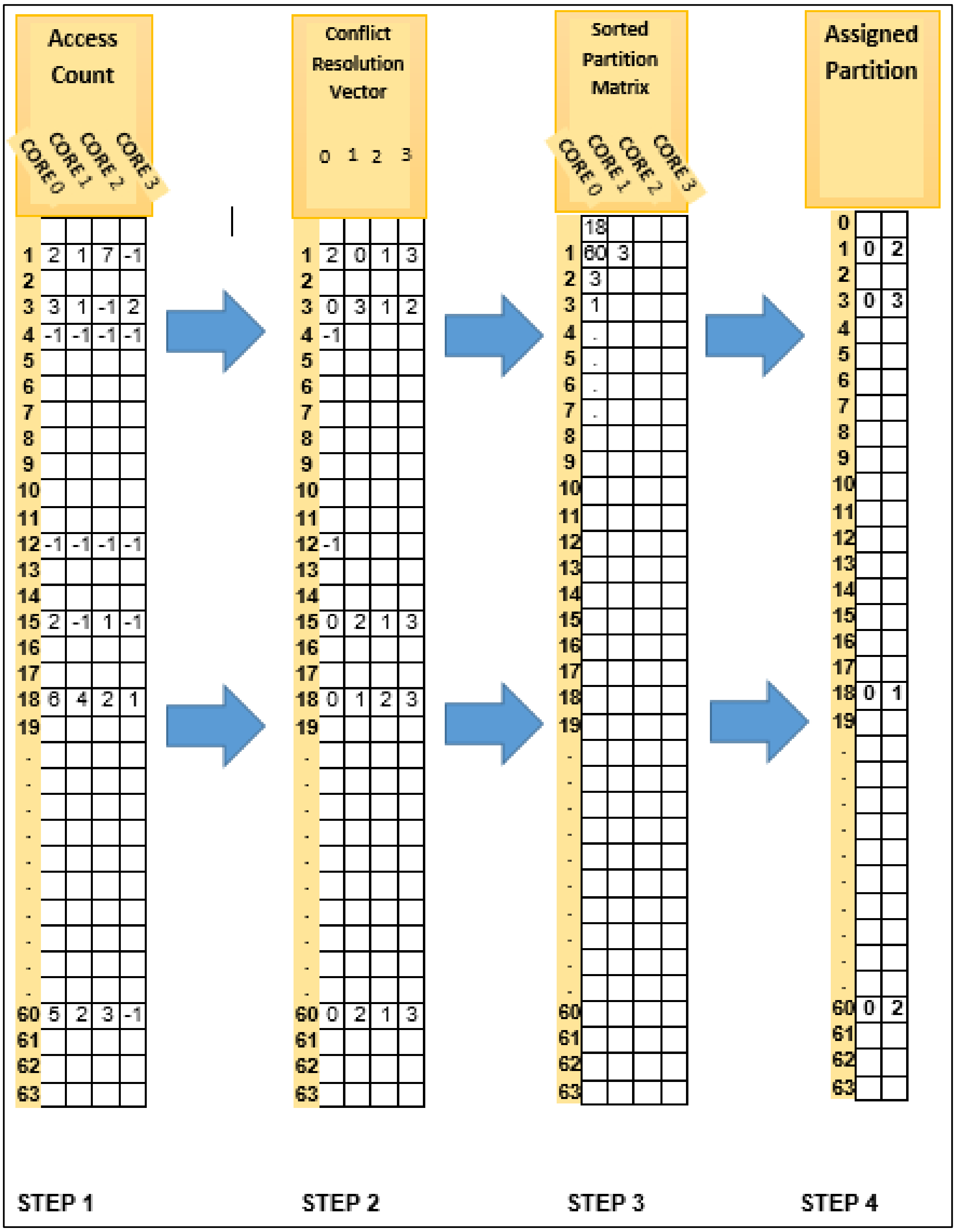
For example, we plan to give 8 cache partitions to three very harmful applications for a combination of 0,2,2,2 classes and give 40 cache partitions to the harmless application.

An important point not shown in the Table 3.1 also need to say, the value of “4” indicates that an application is classified as “No Operation”. In this case, we do not give any partitions to this application.

After classifying the applications, we assigned partitions to each application. So, we use the Partitioner algorithm for realizing this step. We use 4 methods in order to implement Partitioner:

1) ConflictResolution, 2) AssignPartitions, 3)SearchConflictResolutionVector

4) ShiftConflictEntries.



STEP 1 — Access Count

	CORE 0	CORE 1	CORE 2	CORE 3
1	2	1	7	-1
2				
3	3	1	-1	2
4	-1	-1	-1	-1
5				
6				
7				
8				
9				
10				
11				
12	-1	-1	-1	-1
13				
14				
15	2	-1	1	-1
16				
17				
18	6	4	2	1
19				
-				
-				
-				
-				
-				
-				
-				
-				
-				
-				
60	5	2	3	-1
61				
62				
63				

STEP 2 — Conflict Resolution Vector

	0	1	2	3
1	2	0	1	3
2				
3	0	3	1	2
4	-1			
5				
6				
7				
8				
9				
10				
11				
12	-1			
13				
14				
15	0	2	1	3
16				
17				
18	0	1	2	3
19				
-				
-				
-				
-				
-				
-				
-				
-				
-				
-				
60	0	2	1	3
61				
62				
63				

STEP 3 — Sorted Partition Matrix

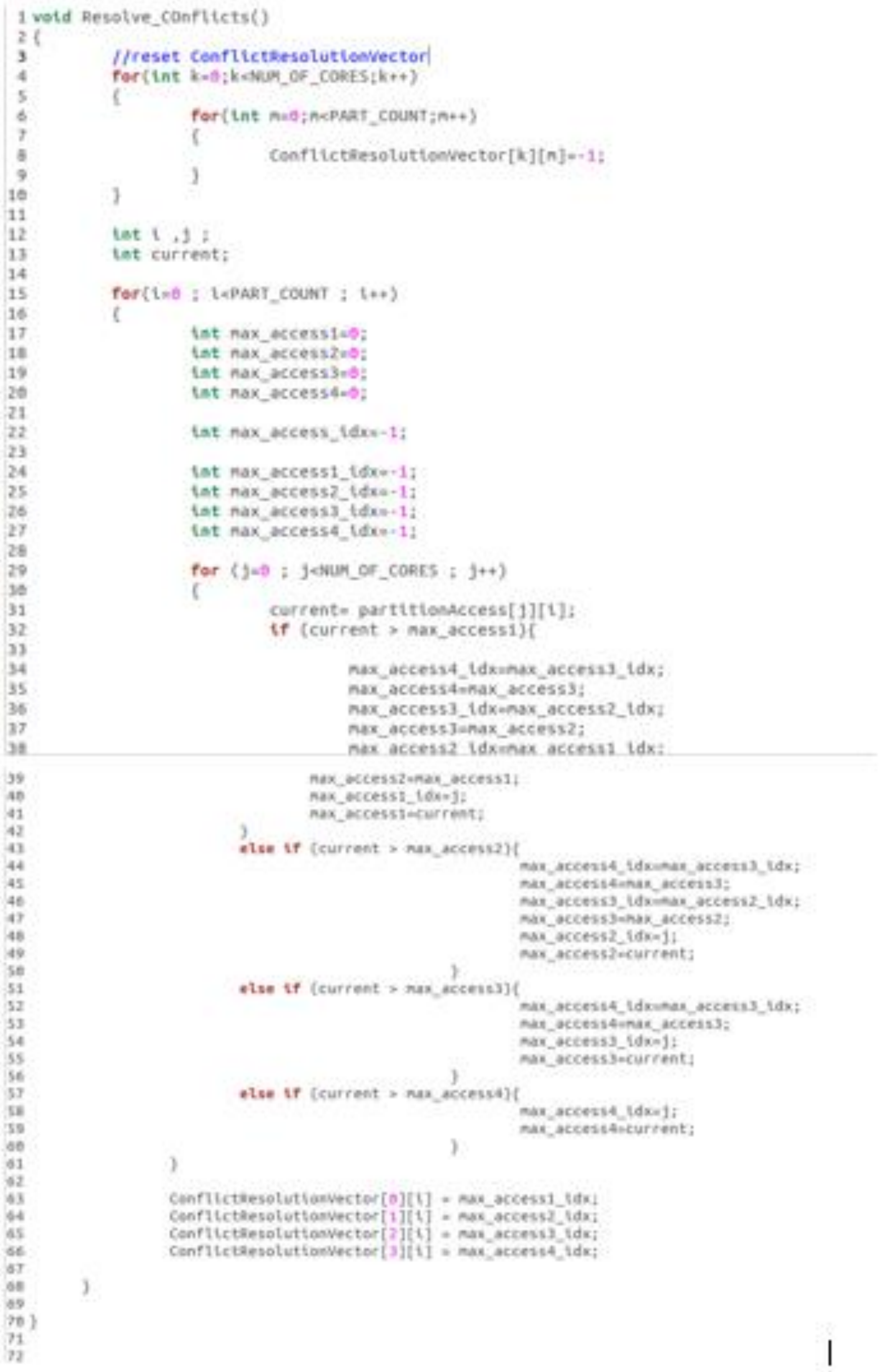
	CORE 0	CORE 1	CORE 2	CORE 3
	18			
1	60	3		
2	3			
3	1			
4	.			
5	.			
6	.			
7	.			
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
-				
-				
-				
-				
-				
-				
-				
-				
-				
-				
60				
61				
62				
63				

STEP 4 — Assigned Partition

0		
1	0	2
2		
3	0	3
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18	0	1
19		
-		
-		
-		
-		
-		
-		
-		
-		
-		
-		
60	0	2
61		
62		
63		

Figure 3.6. Steps of partitioning algorithm

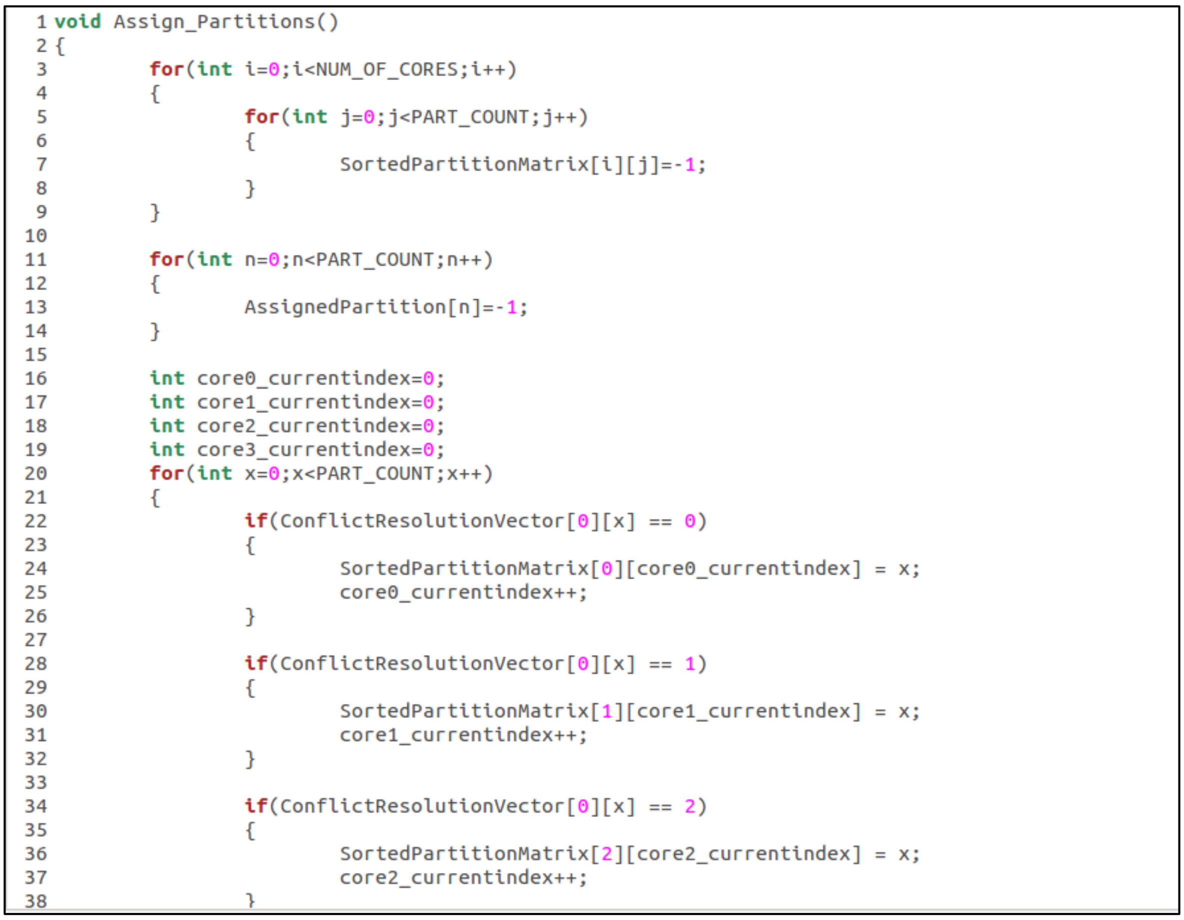
As seen in Figure 3.6, we have a matrix which is called Access Count, this matrix keeps the number of access of cores. In Conflict Resolution Vector, we sort the number of access of each core. To do this, we use the Resolve Conflicts method, as you see in the figure 3.7.



```
void Resolve_COnflicts()
{
        //reset ConflictResolutionVector
        for(int k=0;k<NUM_OF_CORES;k++)
        {
                for(int m=0;m<PART_COUNT;m++)
                {
                        ConflictResolutionVector[k][m]=-1;
                }
        }

        int i ,j ;
        int current;

        for(i=0 ; i<PART_COUNT ; i++)
        {
                int max_access1=0;
                int max_access2=0;
                int max_access3=0;
                int max_access4=0;

                int max_access_idx=-1;

                int max_access1_idx=-1;
                int max_access2_idx=-1;
                int max_access3_idx=-1;
                int max_access4_idx=-1;

                for (j=0 ; j<NUM_OF_CORES ; j++)
                {
                        current= partitionAccess[j][i];
                        if (current > max_access1){

                                max_access4_idx=max_access3_idx;
                                max_access4=max_access3;
                                max_access3_idx=max_access2_idx;
                                max_access3=max_access2;
                                max_access2_idx=max_access1_idx;
                                max_access2=max_access1;
                                max_access1_idx=j;
                                max_access1=current;
                        }
                        else if (current > max_access2){
                                                        max_access4_idx=max_access3_idx;
                                                        max_access4=max_access3;
                                                        max_access3_idx=max_access2_idx;
                                                        max_access3=max_access2;
                                                        max_access2_idx=j;
                                                        max_access2=current;
                                                }
                        else if (current > max_access3){
                                                        max_access4_idx=max_access3_idx;
                                                        max_access4=max_access3;
                                                        max_access3_idx=j;
                                                        max_access3=current;
                                                }
                        else if (current > max_access4){
                                                        max_access4_idx=j;
                                                        max_access4=current;
                                                }
                }

                ConflictResolutionVector[0][i] = max_access1_idx;
                ConflictResolutionVector[1][i] = max_access2_idx;
                ConflictResolutionVector[2][i] = max_access3_idx;
                ConflictResolutionVector[3][i] = max_access4_idx;

        }

}
```

Figure 3.7. Conflict Resolution Algorithm

We defined some variables, then we set the value of partitionAccess[j][i] to current. After that, we compare the number of access of cores and current. According to this comparison, we sort the cores in Conflict Resolution array. For example, when we want to know which core has the most accesses to partition 18, we need to look "ConflictResolutionVector[0][18]". For the second largest number of access for part 18, we need to look "ConflictResolutionVector[1][18]" and so on. At the end, Conflict Resolution vector says us, which core is the most access to the each part of LLC.



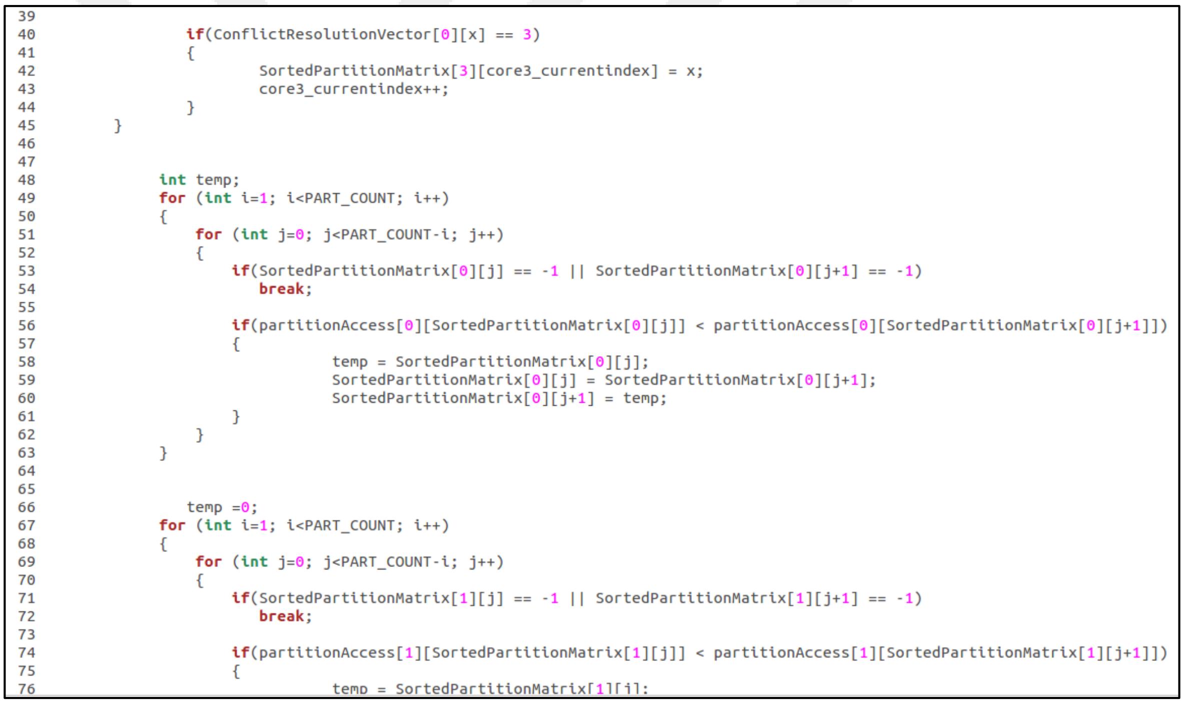
```
void Assign_Partitions()
{
        for(int i=0;i<NUM_OF_CORES;i++)
        {
                for(int j=0;j<PART_COUNT;j++)
                {
                        SortedPartitionMatrix[i][j]=-1;
                }
        }

        for(int n=0;n<PART_COUNT;n++)
        {
                AssignedPartition[n]=-1;
        }

        int core0_currentindex=0;
        int core1_currentindex=0;
        int core2_currentindex=0;
        int core3_currentindex=0;
        for(int x=0;x<PART_COUNT;x++)
        {
                if(ConflictResolutionVector[0][x] == 0)
                {
                        SortedPartitionMatrix[0][core0_currentindex] = x;
                        core0_currentindex++;
                }

                if(ConflictResolutionVector[0][x] == 1)
                {
                        SortedPartitionMatrix[1][core1_currentindex] = x;
                        core1_currentindex++;
                }

                if(ConflictResolutionVector[0][x] == 2)
                {
                        SortedPartitionMatrix[2][core2_currentindex] = x;
                        core2_currentindex++;
                }
```

Figure 3.8. AssignPartition Algorithm - 1

After creating Conflict Resolution Vector, now we defined the AssignPartition method which is used to sort cores with the helping of ConflictResolutionMatrix.(Figure 3.8.)

In this algorithm, first we used the matrix which is called SortedPartitionMatrix. This matrix sorts from largest to smallest according to the number of access of cores. At the end, Sorted Partition Matrix says us, where core 0 has the most access. This matrix gives the answer to which part is the most accessed by core 0. Likewise for core 1, core 2 and core 3. When we want to know which LLC part is the most accessed by core 0, we need to look SortedPartitionMatrix[0][0] and the second largest access by core 0 is hold SortedPartitionMatrix[0][1].

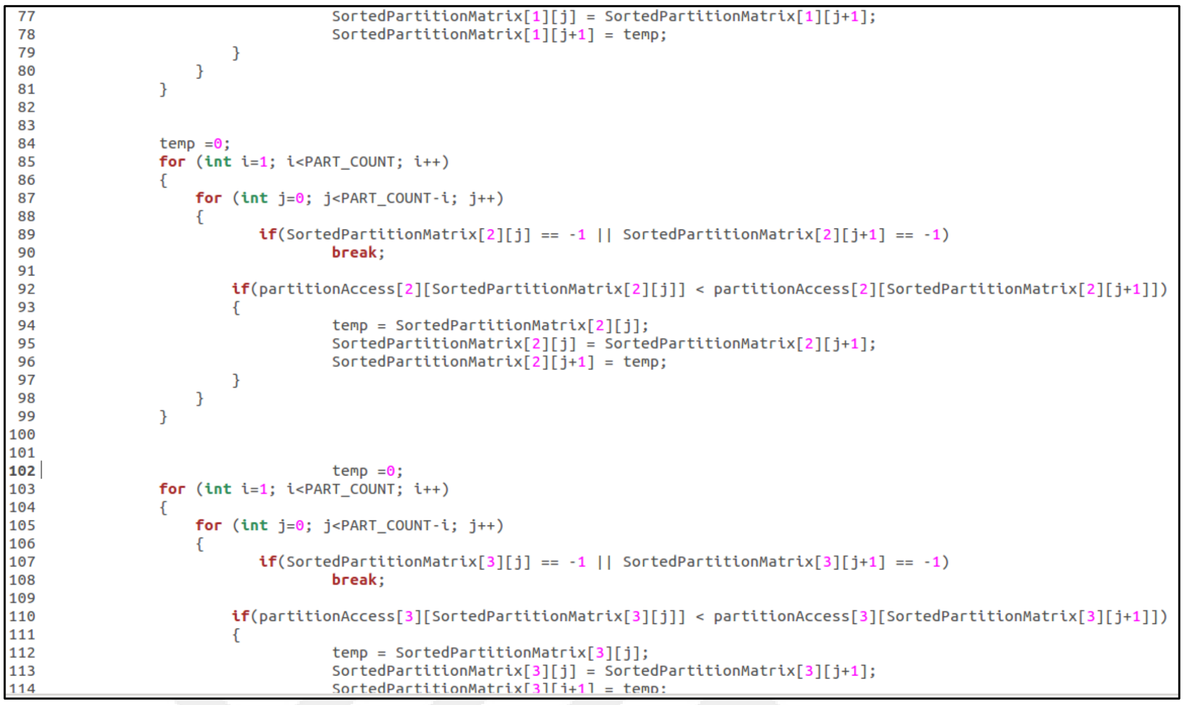
In this algorithm, as seen in Figure 3.8, we firstly set the SortedPartitionMatrix to -1. Then, we transferred the elements of ConflictResolutionMatrix to SortedPartitionMatrix with “for” and “If” statements. (as you see Figure 3.8, 3.9, 3.10 and 3.11)



```

                if(ConflictResolutionVector[0][x] == 3)
                {
                        SortedPartitionMatrix[3][core3_currentindex] = x;
                        core3_currentindex++;
                }
        }


             int temp;
             for (int i=1; i<PART_COUNT; i++)
             {
                 for (int j=0; j<PART_COUNT-i; j++)
                 {
                     if(SortedPartitionMatrix[0][j] == -1 || SortedPartitionMatrix[0][j+1] == -1)
                        break;

                     if(partitionAccess[0][SortedPartitionMatrix[0][j]] < partitionAccess[0][SortedPartitionMatrix[0][j+1]])
                     {
                                temp = SortedPartitionMatrix[0][j];
                                SortedPartitionMatrix[0][j] = SortedPartitionMatrix[0][j+1];
                                SortedPartitionMatrix[0][j+1] = temp;
                     }
                 }
             }


                temp =0;
             for (int i=1; i<PART_COUNT; i++)
             {
                 for (int j=0; j<PART_COUNT-i; j++)
                 {
                     if(SortedPartitionMatrix[1][j] == -1 || SortedPartitionMatrix[1][j+1] == -1)
                        break;

                     if(partitionAccess[1][SortedPartitionMatrix[1][j]] < partitionAccess[1][SortedPartitionMatrix[1][j+1]])
                     {
                                temp = SortedPartitionMatrix[1][j];
```

Figure 3.9. AssignPartition Algorithm – 2

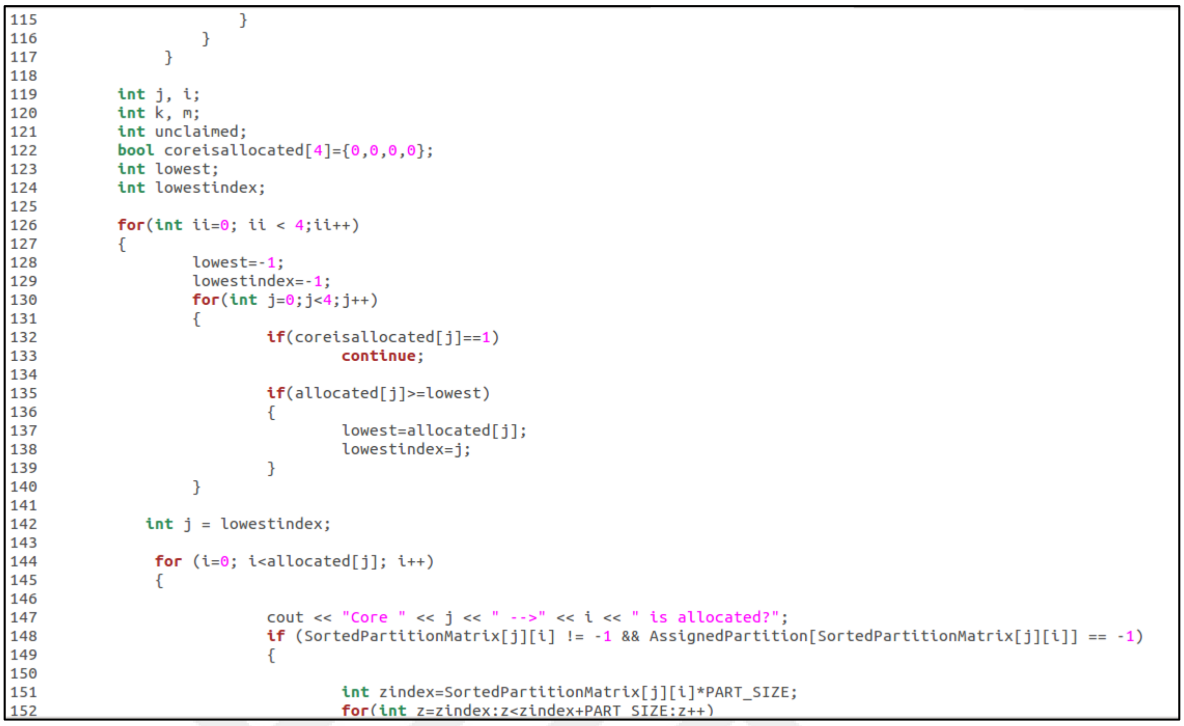
After transferring, as seen in Figure 3.9 and 3.10, we sort the numbers from larger to smaller for each core. In Figure 3.9 and 3.10, we can see easily sorted array.



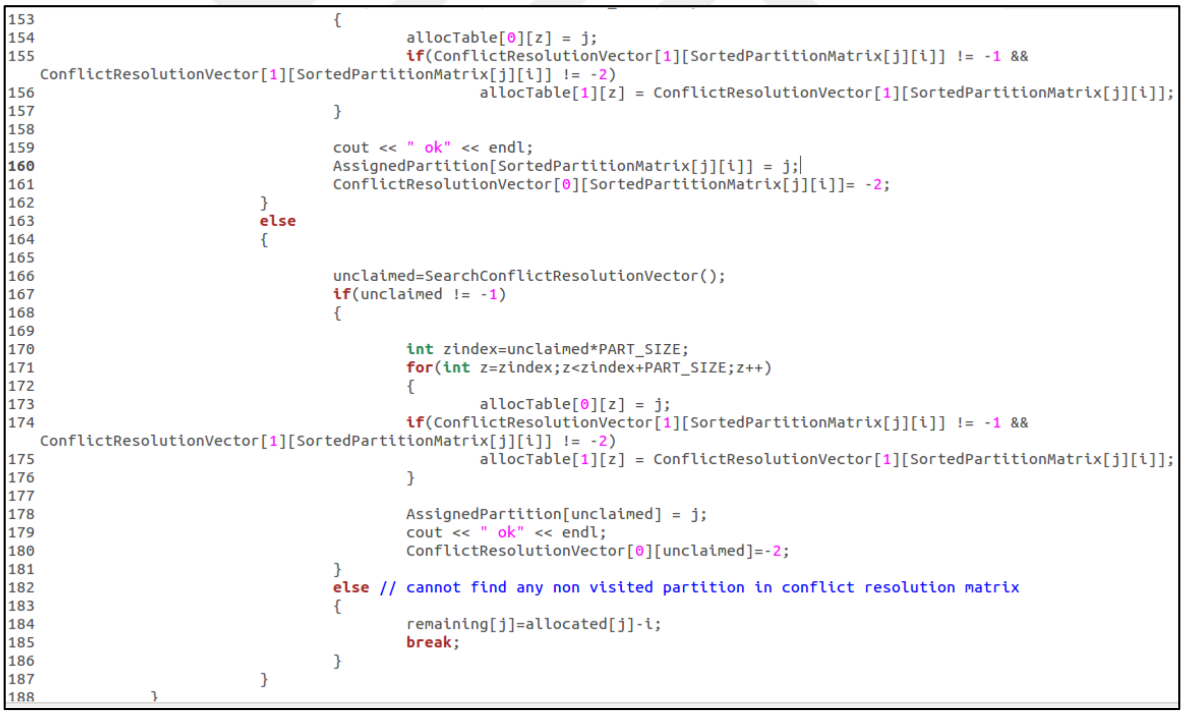
```
                   SortedPartitionMatrix[1][j] = SortedPartitionMatrix[1][j+1];
                   SortedPartitionMatrix[1][j+1] = temp;
        }
    }
}


temp =0;
for (int i=1; i<PART_COUNT; i++)
{
    for (int j=0; j<PART_COUNT-i; j++)
    {
           if(SortedPartitionMatrix[2][j] == -1 || SortedPartitionMatrix[2][j+1] == -1)
                   break;

        if(partitionAccess[2][SortedPartitionMatrix[2][j]] < partitionAccess[2][SortedPartitionMatrix[2][j+1]])
        {
                   temp = SortedPartitionMatrix[2][j];
                   SortedPartitionMatrix[2][j] = SortedPartitionMatrix[2][j+1];
                   SortedPartitionMatrix[2][j+1] = temp;
        }
    }
}


                   temp =0;
for (int i=1; i<PART_COUNT; i++)
{
    for (int j=0; j<PART_COUNT-i; j++)
    {
           if(SortedPartitionMatrix[3][j] == -1 || SortedPartitionMatrix[3][j+1] == -1)
                   break;

        if(partitionAccess[3][SortedPartitionMatrix[3][j]] < partitionAccess[3][SortedPartitionMatrix[3][j+1]])
        {
                   temp = SortedPartitionMatrix[3][j];
                   SortedPartitionMatrix[3][j] = SortedPartitionMatrix[3][j+1];
                   SortedPartitionMatrix[3][j+1] = temp;
```

Figure 3.10. AssignPartition Algorithm – 3

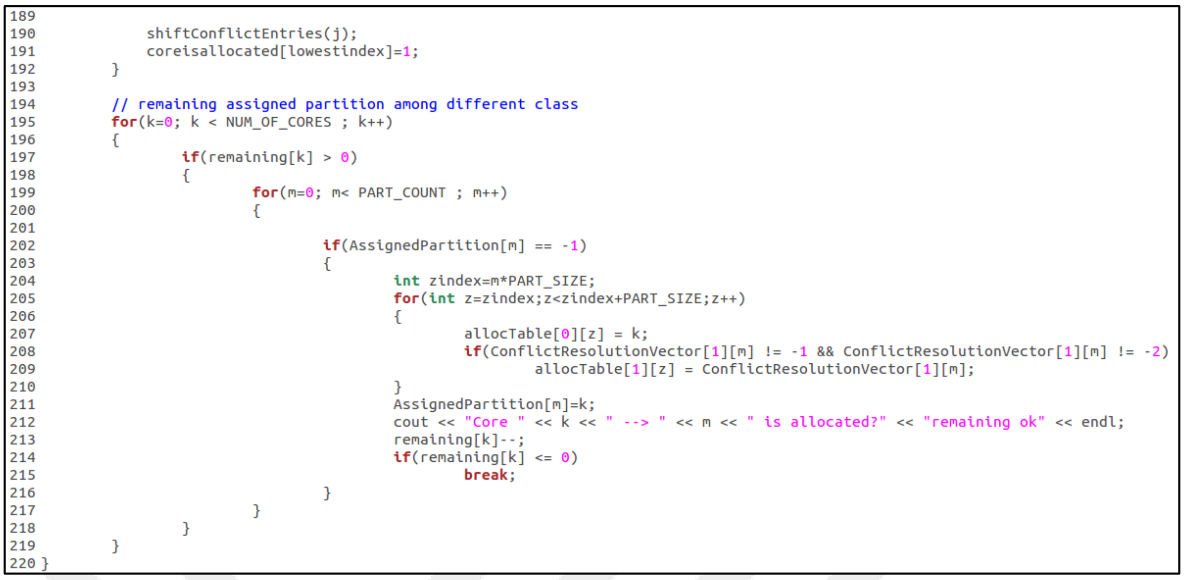
Then, we place the core in the Assign Partition array. This array is, as you see in the figure 3.6, two dimentional array. In this design, instead of giving a whole partition to a single application, we plan to allocate it to two applications. We share a partition among two cores. To do this, we write a code in Figure 3.11. To be fair, we decided to start lowest partition to place in the array. Firstly, we look the allocated array which keeps the number of partition of cores, then we choose the lowest one then search the number of core which is the number of core in the SortedPartitionMatrix in the AssignedPartition array. After, we placed it in the array then we set -2 to the number of core in the ConflictResolutionVector matrix.



```
                     }
                 }
             }

        int j, i;
        int k, m;
        int unclaimed;
        bool coreisallocated[4]={0,0,0,0};
        int lowest;
        int lowestindex;

        for(int ii=0; ii < 4;ii++)
        {
                lowest=-1;
                lowestindex=-1;
                for(int j=0;j<4;j++)
                {
                        if(coreisallocated[j]==1)
                                continue;

                        if(allocated[j]>=lowest)
                        {
                                lowest=allocated[j];
                                lowestindex=j;
                        }
                }

           int j = lowestindex;

            for (i=0; i<allocated[j]; i++)
            {

                        cout << "Core " << j << " -->" << i << " is allocated?";
                        if (SortedPartitionMatrix[j][i] != -1 && AssignedPartition[SortedPartitionMatrix[j][i]] == -1)
                        {

                                int zindex=SortedPartitionMatrix[j][i]*PART_SIZE;
                                for(int z=zindex;z<zindex+PART_SIZE;z++)
```



```
                                {
                                        allocTable[0][z] = j;
                                        if(ConflictResolutionVector[1][SortedPartitionMatrix[j][i]] != -1 && ConflictResolutionVector[1][SortedPartitionMatrix[j][i]] != -2)
                                                allocTable[1][z] = ConflictResolutionVector[1][SortedPartitionMatrix[j][i]];
                                }

                                cout << " ok" << endl;
                                AssignedPartition[SortedPartitionMatrix[j][i]] = j;
                                ConflictResolutionVector[0][SortedPartitionMatrix[j][i]]= -2;
                        }
                        else
                        {

                                unclaimed=SearchConflictResolutionVector();
                                if(unclaimed != -1)
                                {

                                        int zindex=unclaimed*PART_SIZE;
                                        for(int z=zindex;z<zindex+PART_SIZE;z++)
                                        {
                                                allocTable[0][z] = j;
                                        if(ConflictResolutionVector[1][SortedPartitionMatrix[j][i]] != -1 && ConflictResolutionVector[1][SortedPartitionMatrix[j][i]] != -2)
                                                allocTable[1][z] = ConflictResolutionVector[1][SortedPartitionMatrix[j][i]];
                                        }

                                        AssignedPartition[unclaimed] = j;
                                        cout << " ok" << endl;
                                        ConflictResolutionVector[0][unclaimed]=-2;
                                }
                                else // cannot find any non visited partition in conflict resolution matrix
                                {
                                        remaining[j]=allocated[j]-i;
                                        break;
                                }
                        }
            }
```

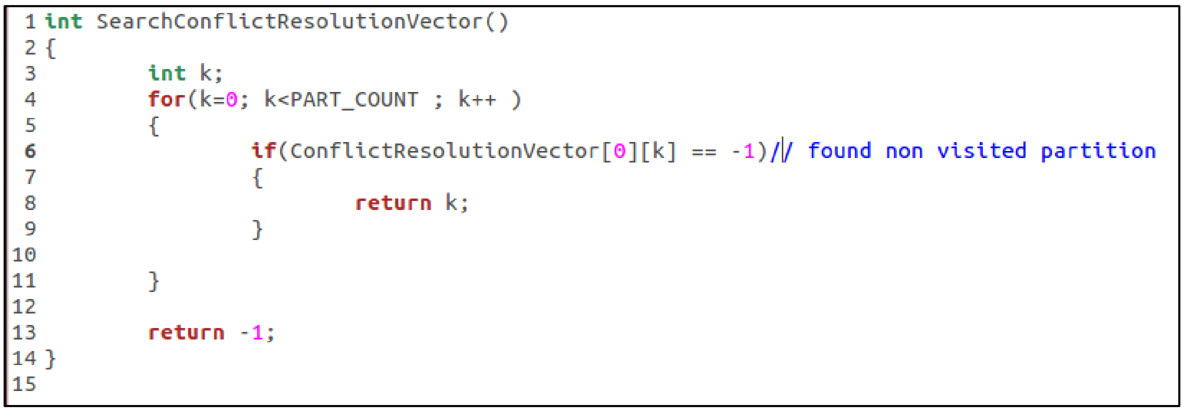


```
            shiftConflictEntries(j);
            coreisallocated[lowestindex]=1;
        }

        // remaining assigned partition among different class
        for(k=0; k < NUM_OF_CORES ; k++)
        {
                if(remaining[k] > 0)
                {
                        for(m=0; m< PART_COUNT ; m++)
                        {

                                if(AssignedPartition[m] == -1)
                                {
                                        int zindex=m*PART_SIZE;
                                        for(int z=zindex;z<zindex+PART_SIZE;z++)
                                        {
                                                allocTable[0][z] = k;
                                                if(ConflictResolutionVector[1][m] != -1 && ConflictResolutionVector[1][m] != -2)
                                                        allocTable[1][z] = ConflictResolutionVector[1][m];
                                        }
                                        AssignedPartition[m]=k;
                                        cout << "Core " << k << " --> " << m << " is allocated?" << "remaining ok" << endl;
                                        remaining[k]--;
                                        if(remaining[k] <= 0)
                                                break;
                                }
                        }
                }
        }
}
```

Figure 3.11. AssignPartition Algorithm – 4

After to set -2 the number of cores in ConflictResolutionVector matrix, we called the method which is called shiftConflictEntries. Then, we keep a remaining array which is used for remaining partition.

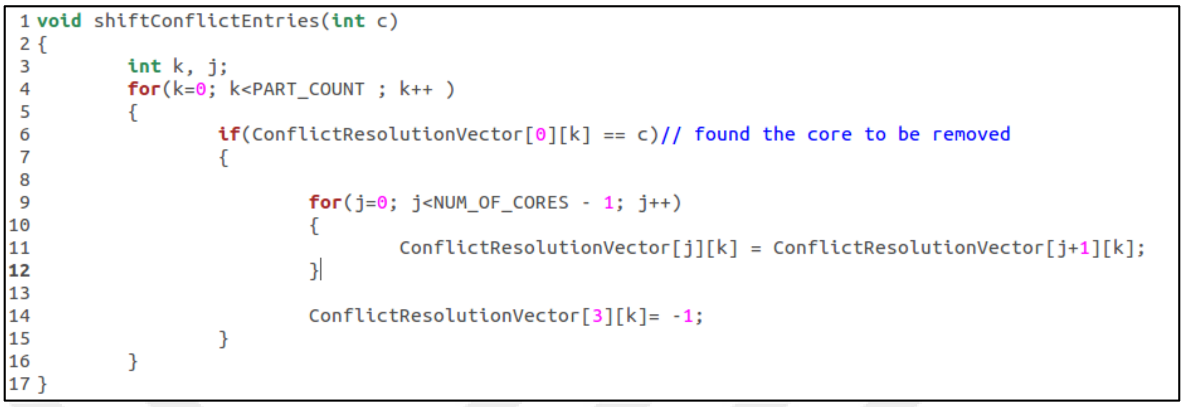
For example, if we assign 6 partitions to SortedPartitionMatrix but we have to assign 8 partition. Then, we use the remaining array for 2 partitions. We scan ConflictResolutionMatrix, if we see -1, then we assigns it to the SortedPartitionMatrix.



```
int SearchConflictResolutionVector()
{
        int k;
        for(k=0; k<PART_COUNT ; k++ )
        {
                if(ConflictResolutionVector[0][k] == -1)// found non visited partition
                {
                        return k;
                }

        }

        return -1;
}

```

Figure 3.12. SearchConflictResolution Algorithm

In SearchConflictResolution algorithm (Figure 3.12), we research the non-found visited partition for helping the AssignPartition algorithm.



```
void shiftConflictEntries(int c)
{
        int k, j;
        for(k=0; k<PART_COUNT ; k++ )
        {
                if(ConflictResolutionVector[0][k] == c)// found the core to be removed
                {

                        for(j=0; j<NUM_OF_CORES - 1; j++)
                        {
                                ConflictResolutionVector[j][k] = ConflictResolutionVector[j+1][k];
                        }

                        ConflictResolutionVector[3][k]= -1;
                }
        }
}
```

Figure 3.13. shiftConflictEntries Algorithm

In shiftConflictEntries Algorithm (Figure 3.13), after we assign the number of core which is in the ConflictResolutionMatrix, we shift the left array.

3.4. HARDWARE IMPLEMENTATION AND COMPLEXITY OVERHEAD

In our implementation, we proposed two additional hardware circuit. In order to estimate the complexity overhead, we calculate the approximated number of transistors for each hardware circuit that is used in our implementation.

One of the issues which we give priority in the project (as we mentioned earlier), we will encounter many more examples in the near future core processor that can run on a scalable system is put forward. At this point, in Figure 3.15 and Figure 3.16 provides details of our knowledge-Cache circuit complexity calculated as a function of the number of cores. As can be seen in the graph in Figure 3.14, for 64-core processor, our designed cache system memory space requirement is only 3% of the area corresponds to the requirements for 4 MB L3 cache. Much bigger than 4MB for 64-core processor, the last level cache sizes to be used shall be at the rate negligible level. The only reason for non-linear increase in this graph, with the number of cores of partitioner table is exponentially increasing space requirements.

64 and older with a processor core instead of in the partitioner table at run-time partitioning determines the number envisage the use of a control circuit.

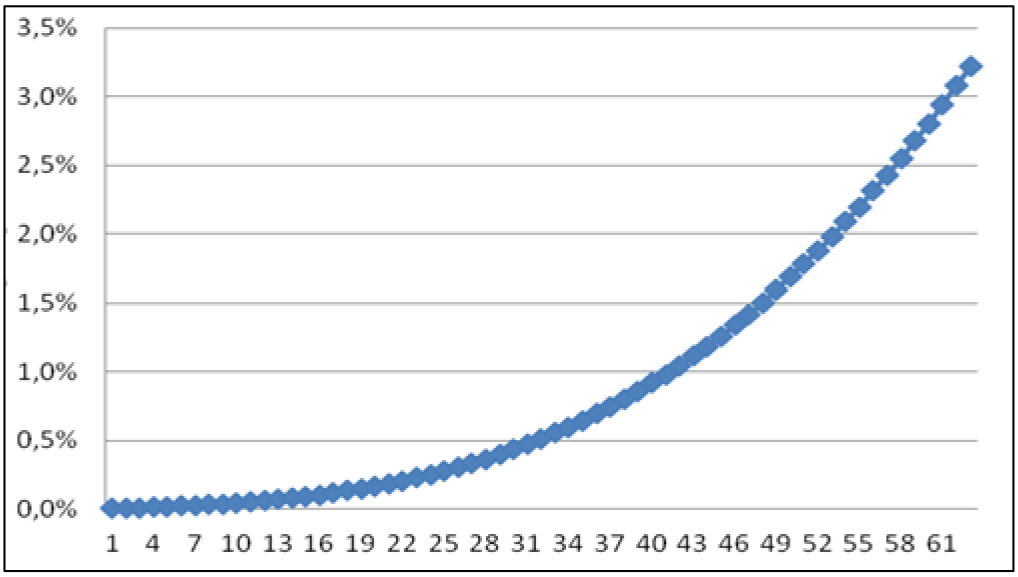


Figure 3.14. According to the L3 cache, the space requirements of the proposed design variation with the number of cores

The hardware implementation of classifier circuit is shown by the figure 3.15. The outputs of this circuit are classes. On the other hand, Figure 3.16 shows us the hardware implementation of partitioner.

Transistor numbers of each hardware circuit in our method implementation with the number of transistors of used cache space are shown in table 3.2.

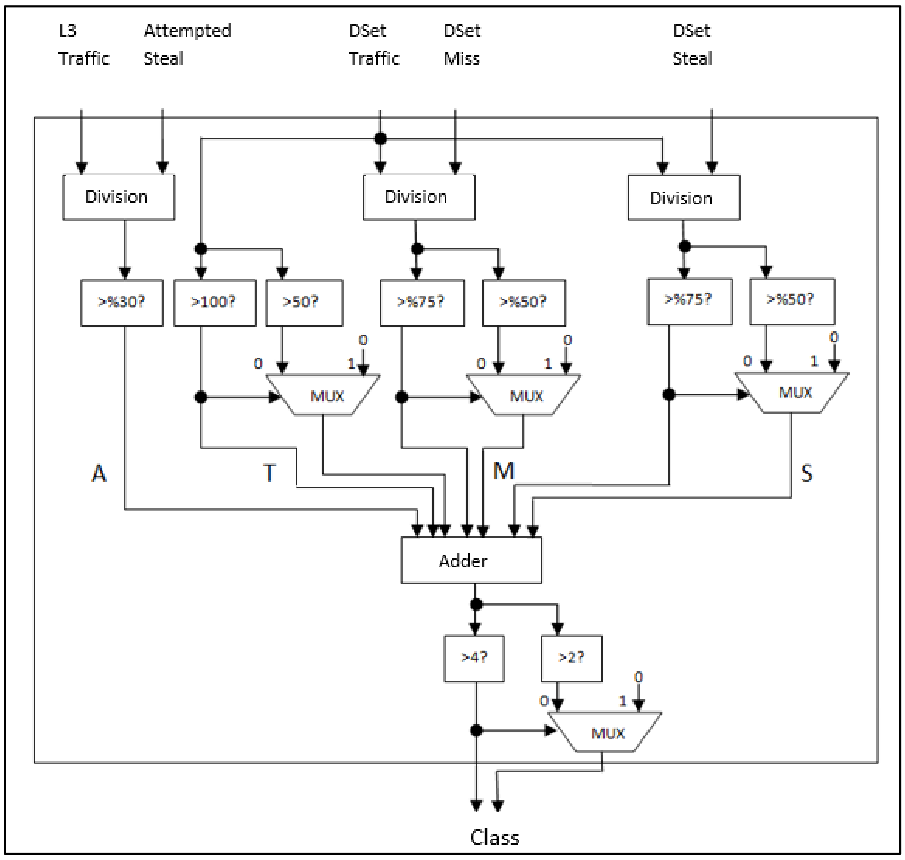


Figure 3.15. Hardware implementation of classifier

Table 3.2. Classifier cost

Classifier elements	**Bit numbers in each element**	**Transistor numbers in each element**
Five counters for L3 Traffic, Attempted Steal, Miss, Steal and DSet	64	(5*64*6) =1920
Three division units	64	10368 = (3*64*54)
Four multiplication unit	20	4320 = (4*20*54)
One adder unit	7	196 = (7*28)
Nine comparator unit	7	1323 = (9*7*21)
Total cost:	18127 transistors	

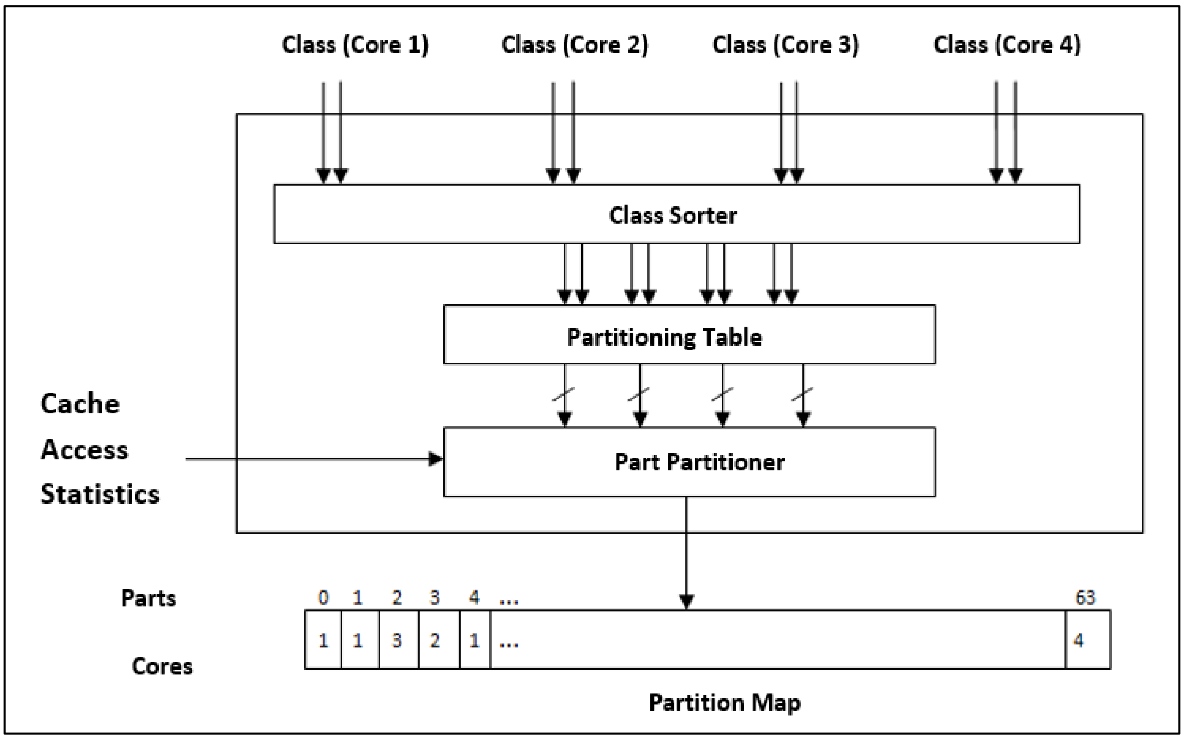


Figure 3.16. Hardware implementation of partitioner

4. TEST ENVIRONMENT

4.1. MACSIM SIMULATOR

Macsim is a heterogeneous architecture simulator [15]. x86 and NVDIA PTX instructions can be simulated by Macsim. It models micro-architectural behavior, including pipeline stages, multi-threading execution and memory systems. It can simulate a variety of architectures which are Intel’s Sandy Bridge and NVDIA’s Fermi. Additionally, homogeneous ISA multicore simulations can be simulated by Macsim.

Our reason to use Macsim is that it can determine the behaviors of various applications and different approaches and algorithms can be compared easily by the help of Macsim. We can change the underlying processor and memory configurations since it provides a fully customizable interface.

4.1.1. MEMORY SYSTEM IN MACSIM

Caches: Every cache structure has storage and multiple queues. In Figure 4.1, the overall structure of a single cache is depicted in Macsim. There are two flows:

- Cache Access Flow: Data flow from processor to upper level cache. In case of a cache miss, cache is accessed through this flow.
- Cache Fill Flow: In case of a cache miss, data is supplied from the lower level cache or DRAM.

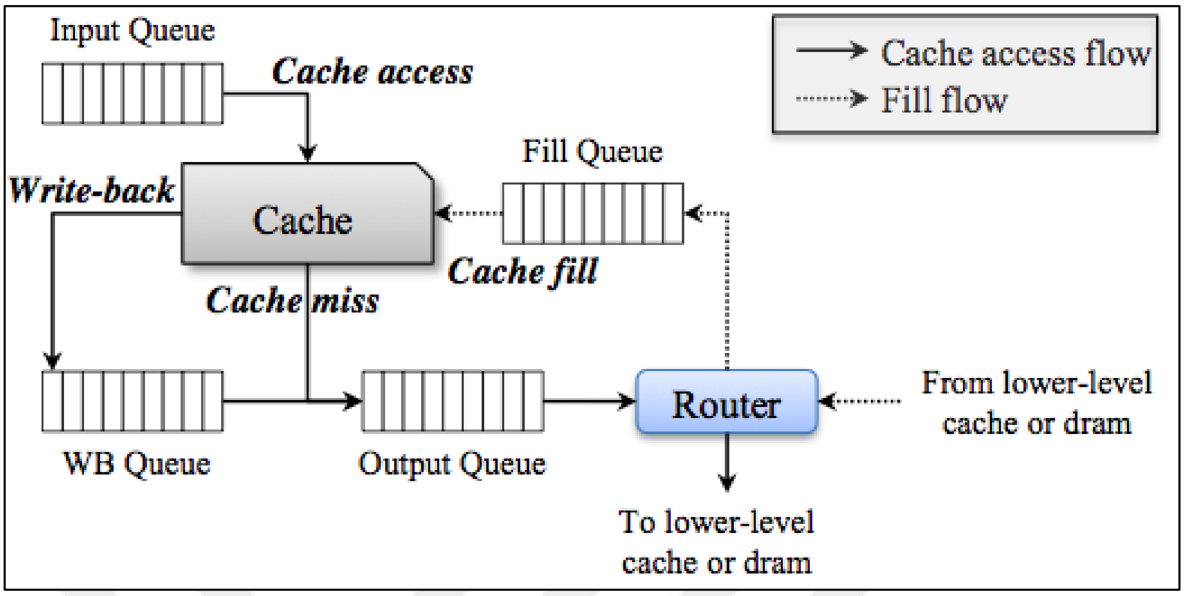


Figure 4.1. Cache Structure of MACSIM

Queues:

- **Input Queue:** Requests are forwarded to the cache triggered by upper-level cache misses are inserted into this queue.
- **Output Queue:** Requests that miss in this cache are inserted into the output queue to be forwarded to a lower-level cache. If no lower-level cache is available, then requests are forward to the main memory.
- **Write-back Queue:** Write-back cache is one of the models of Macsim. When a dirty cache line is evicted, the line must be written back into the next level cache.
- **Fill Queue:** Data returned from the next level cache or main memory is inserted into the fill queue.

4.1.2. HIERARCHY

Macsim is a flexible simulator so it can work with different memory hierarchies. Each level can be configured independently of other levels in the cache hierarchy. In the following figure, we can see the base memory hierarchy of Macsim without DRAM memory. (Figure 4.2)

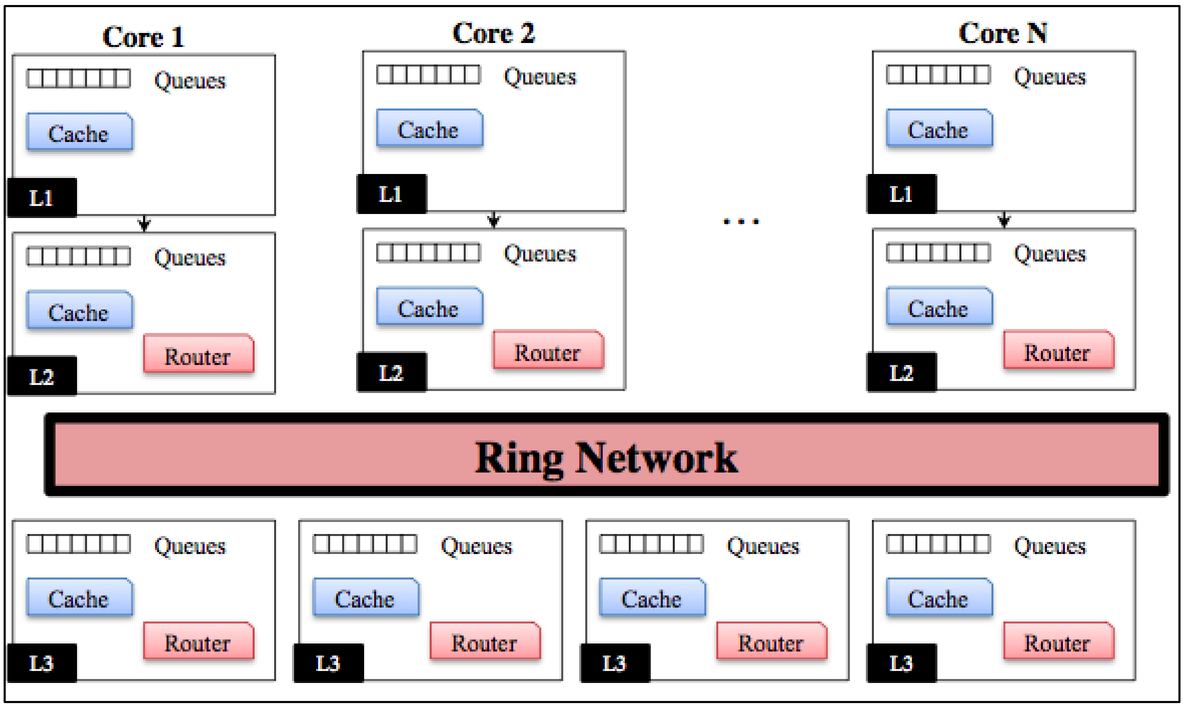


Figure 4.2. Memory System of MacSim

- Cache has 3 levels which are L1, L2 and L3.
- If required, the caches can be connected to each other.
- Each core has unique L1 and L2 caches.
- When required, the local router is enabled within a cache structure.
- All cores share the L3 cache. But, address regions are statically partitioned and also each tile is responsible for sub-regions.

4.2.BENCHMARKS

The SPEC CPU 2006 [25] benchmark is SPEC's next-generation, industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler.

This benchmark suite includes the SPECint benchmarks and the SPECfp benchmarks. The SPECint 2006 benchmark contains 12 different benchmark tests and the SPECfp 2006 benchmark contains 19 different benchmark tests. We use some of these benchmarks mixed, here are the descriptions of the applications that are used in our work, shown below:

- **401.bzip2:** Performs no file I/O other than reading the input. All compression and decompression happens entirely in memory.

- **403.gcc:** Based on gcc 3.2, it generates code for an AMD Opteron processor. The benchmark runs as a compiler with many of its optimization flags enabled. It has its inlining heuristics altered slightly, so as to inline more code than would be typical on a Unix system in 2002. It is expected that this effect will be more typical of compiler usage in 2006. This is to make 403.gcc spend more time analyzing its source code inputs and use more memory.

- **429.mcf:** It is derived from MCF, a program used for single-depot vehicle scheduling in public mass transportation. The program is designed for the solution of single-depot scheduling problems planning transportation. It considers one single depot and a homogenous vehicle fleet.It is the task to schedule all timetabled trips to so-called blocks. The network simplex algorithm is a specialized version of the well-known simplex algorithm for network flow problems. The main work of our network simplex implementation is pointer and integer arithmetic.

- **445.gobmk:** This program plays Go[1] and executes a set of commands to analyze Go positions.

- **462.libquantum:** This is a library for the simulation of a quantum computer. Quantum computers are based on the principles of quantum mechanics and can solve certain computationally hard tasks in polynomial time.

- **464.h264ref:** This is a reference implementation of H264/AVC (Advanced Video Coding), the latest video compression standard. It replaces the current MPEG-2 standard, for applications such as next-generation DVDs and video broadcasting.

- **473.astar:** This is derived from a portable 2D path-finding library that is used in a game's AI. It implements three different path-finding algorithms.

- **444.namd:** This is derived from the data layout and inner loop of NAMD, a parallel program for the simulation of large bio molecular systems.

- **470.lbm:** This program implements the so-called "Lattice Boltzmann Method" to simulate incompressible fluids. It is the computationally important part of a larger code-use in material science to simulate fluids with free surfaces, in particular the formation and movement of gas bubbles in metal foams. [18]

These benchmarks are provided as source code and require the user to be comfortable using compiler commands as well as other commands via a command interpreter using a console or command prompt window in order to generate executable binaries.

4.3. TRACE GENERATION

For simulations using MacSim, x86 traces are generated using Pin and PTX traces are generated using GPUOcelot. (Figure 4.2.) Internally, MacSim converts both x86 and PTX trace instructions into RISC style micro-ops (uop) which are simulated. In the figure below shows a high-level picture of the operation of the simulator.

[1] Go is a board game for two players. The object is to surround and capture opponent's counters. More info can be found at http://www.britgo.org/about/index.html .

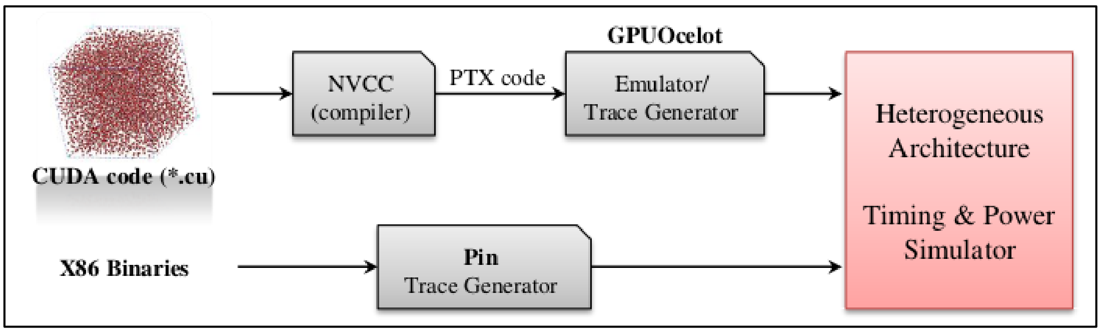


Figure 4.3.The Overview of MACSIM Simulator

In order to generate executable binaries, we use Pin tool 2.12. Pin is a tool for the instrumentation of programs. It supports Linux and Windows executables for IA-32, Intel 64, and IA-64 architectures.

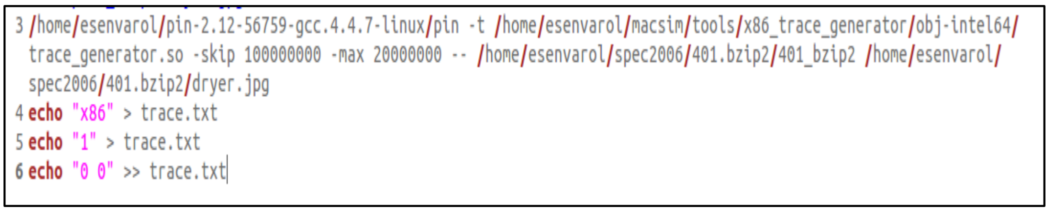
Pin allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The code is added dynamically while the executable is running. This also makes it possible to attach Pin to an already running process.

MacSim includes a CPU (x86) trace generator which is based on Pin [10], a binary instrumentation tool. After installing Pin 1 , the x86 trace generator module has to be built. The command for doing so is:

cd toos/x86_trace_generator

make

This will generate trace_generator.so in the tools/x86_trace_generator/obj-intel64 directory. x86 traces for MacSim can then be generated by running Pin with the generated module. We use generally this script for generate trace:

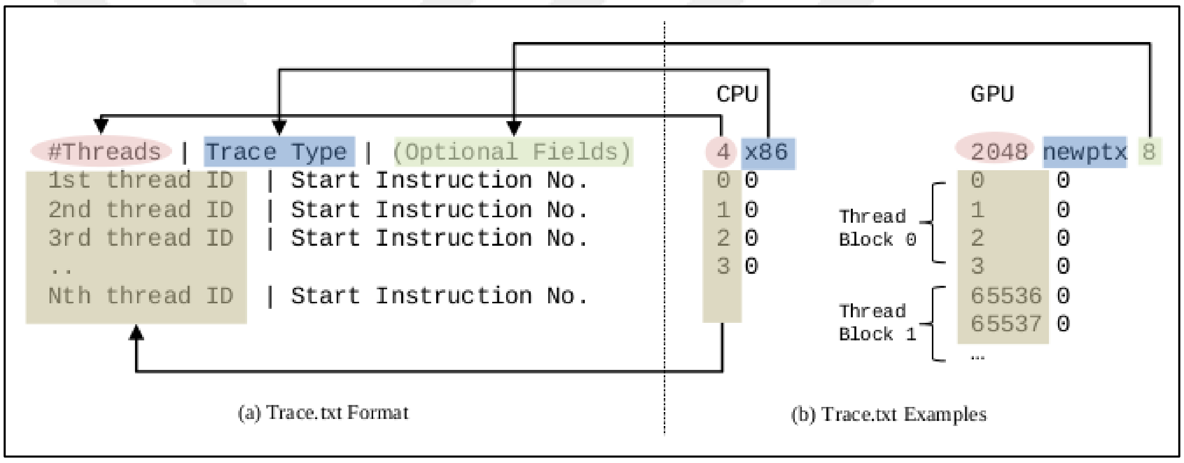


```
/home/esenvarol/pin-2.12-56759-gcc.4.4.7-linux/pin -t /home/esenvarol/macsim/tools/x86_trace_generator/obj-intel64/
 trace_generator.so -skip 100000000 -max 20000000 -- /home/esenvarol/spec2006/401.bzip2/401_bzip2 /home/esenvarol/
 spec2006/401.bzip2/dryer.jpg
echo "x86" > trace.txt
echo "1" > trace.txt
echo "0 0" >> trace.txt
```

Figure 4.4. Trace Generation Script

In the script above, we skip 100.000.000 instructions and limit to 20.000.000 maximum. The trace generator generates two files (in case of a single threaded application) “Trace.txt” and “trace_0.raw”, in the current directory.

- **Trace.txt (info trace):** Contains information about the generated trace files (#threads, trace type, ...).
- **Trace_xx.raw (raw trace):** Contains instruction trace for a thread and is generated for each thread.



CPU
GPU
#Threads | Trace Type | (Optional Fields)
1st thread ID | Start Instruction No.
2nd thread ID | Start Instruction No.
3rd thread ID | Start Instruction No.
..
Nth thread ID | Start Instruction No.
4 x86
0 0
1 0
2 0
3 0
2048 newptx 8
Thread Block 0
0 0
1 0
2 0
3 0
Thread Block 1
65536 0
65537 0
...
(a) Trace.txt Format
(b) Trace.txt Examples

Figure 4.5. Trace.txt format

Figure 4.5 shows the format of Trace.txt and its CPU and GPU examples. As shown in Figure 4.5-(a), the first line in Trace.txt has different fields from the rest of the lines.

- #Threads: indicates the number of threads for which traces have been generated, and this value is equal to the number of lines in the file excluding the first line.
- Trace Type: indicates whether the generated traces are for an x86 application or a PTX kernel.
- Optional Field(s): currently used for PTX traces only and indicates the number of thread blocks that can be assigned to a streaming multiprocessor(SM) core (occupancy).

From the second line onwards, there are two fields in each line: thread id and start instruction number. For each thread, there is a Trace_<thread_id>.raw file which contains the dynamic instruction trace for the thread. Finally, start instruction number indicates when each thread should be started in terms of the number of instructions simulated for the main thread of the application. In a PTX kernel since all warps are ready for execution at the launch of the kernel, the start instruction number for all threads is zero. On the other hand, for a x86 application, the start instruction is non-zero for all threads except thread 0, which is the main (or parent) thread in the application. This is because in most multi-threaded CPU applications, main thread (thread id 0) spawns children threads.

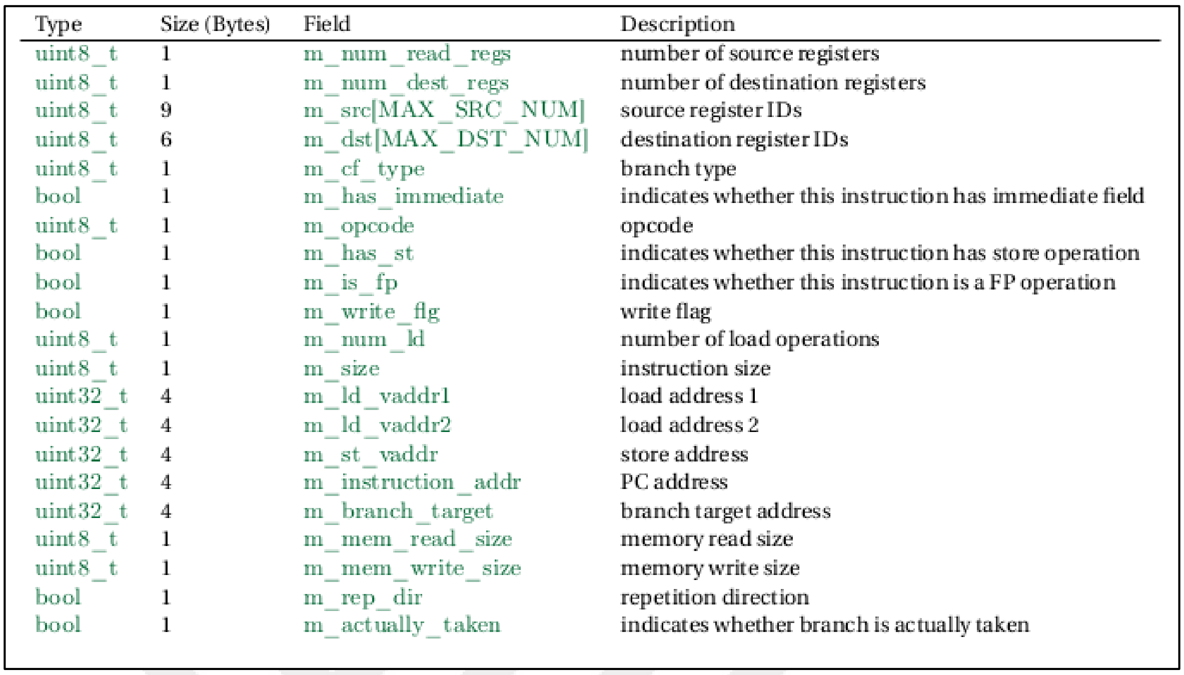
In Figure 2-(b), the CPU trace has four threads and its type is set to x86. The ids of the threads are 0-3 with the corresponding trace files being Trace_0.raw−Trace_3.raw. Thread 0 is ready at the start of simulation, while Threads 1, 2 and 3 become ready when Thread 0 has fetched x, y and z instructions respectively.

In the GPU example, the number of traces files is 2048 since #Threads (representing #Warps in case of GPUs) is 2048. The optional field indicates that eight thread blocks can be assigned to a SM core.

For GPU traces, the id in the file encodes thread block information as well. The warp id and thread block id can be decoded from this id as follows:

```
warp_id = id % (1 << 16)
block_id = id / (1 << 16)
```

Trace_xx.raw is generated for each thread/warp and contains the dynamic instruction trace for the thread/warp in the binary format. The structure/format for encoding instructions is the same in both x86 and PTX traces and looks as Figure 4.6 (in order):



Type	Size (Bytes)	Field	Description
uint8_t	1	m_num_read_regs	number of source registers
uint8_t	1	m_num_dest_regs	number of destination registers
uint8_t	9	m_src[MAX_SRC_NUM]	source register IDs
uint8_t	6	m_dst[MAX_DST_NUM]	destination register IDs
uint8_t	1	m_cf_type	branch type
bool	1	m_has_immediate	indicates whether this instruction has immediate field
uint8_t	1	m_opcode	opcode
bool	1	m_has_st	indicates whether this instruction has store operation
bool	1	m_is_fp	indicates whether this instruction is a FP operation
bool	1	m_write_flg	write flag
uint8_t	1	m_num_ld	number of load operations
uint8_t	1	m_size	instruction size
uint32_t	4	m_ld_vaddr1	load address 1
uint32_t	4	m_ld_vaddr2	load address 2
uint32_t	4	m_st_vaddr	store address
uint32_t	4	m_instruction_addr	PC address
uint32_t	4	m_branch_target	branch target address
uint8_t	1	m_mem_read_size	memory read size
uint8_t	1	m_mem_write_size	memory write size
bool	1	m_rep_dir	repetition direction
bool	1	m_actually_taken	indicates whether branch is actually taken

Figure 4.6. The Structure/Format for Encoding Instructions

Note that the raw trace is compressed with zlib to reduce the sizes of the generated trace files, and the size of each field is the size before the compression.

5. EXPERIMENTAL RESULTS

In order to evaluate our proposed design we used Macsim, a heterogeneous architecture simulator, which is trace-driven and cycle-level. In our study, we simulated four or eight cores with identical specifications. We only change the cache parameters throughout our experimental study in "params.in" file. Here is the experimental setup, show in Table 5.1.

Table 5.1. Experimental Setup Table

Parametre	Configuration
L1 I-Cache (instruction)	64 KB, 64-sets, 128B block, 8-Way, LRU Replacement
L1 D-Cache (data)	64 KB, 128-sets, 64B block, 8- Way, LRU Replacement
L2Cache (instruction+data)	512 KB, 1024-sets, 64B block, 8- Way, LRU Replacement
L3 Cache (shared)	1 MB, 1024-sets, 64B block, 16- Way, 1-tile, LRU Replacement

In the figures given below, we compare the results of our proposed cache architecture with the results of baseline (uncontrolled shared cache). First, we executed four applications concurrently and evaluated throughput value of each applications. Here are the results of these study shown below:

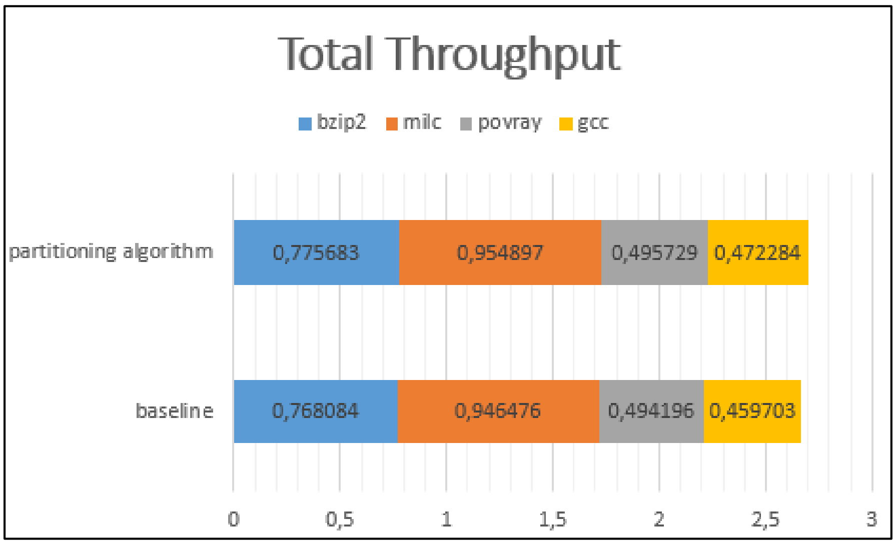


Figure 5.1. results of bzip2, milc, povray and gcc

In the figure 5.1, we executed bzip2, milc, povray and gcc traces concurrently. Bzip2 and povray are considered to be harmful application by our classifier algorithm. On the other hand, milc and gcc are considered to be very harmful. According to the classification, our partition algorithm gave more cache resources to harmful applications than very harmful application. At the end, when we compared to baseline, the total throughput increase is approximately %2.

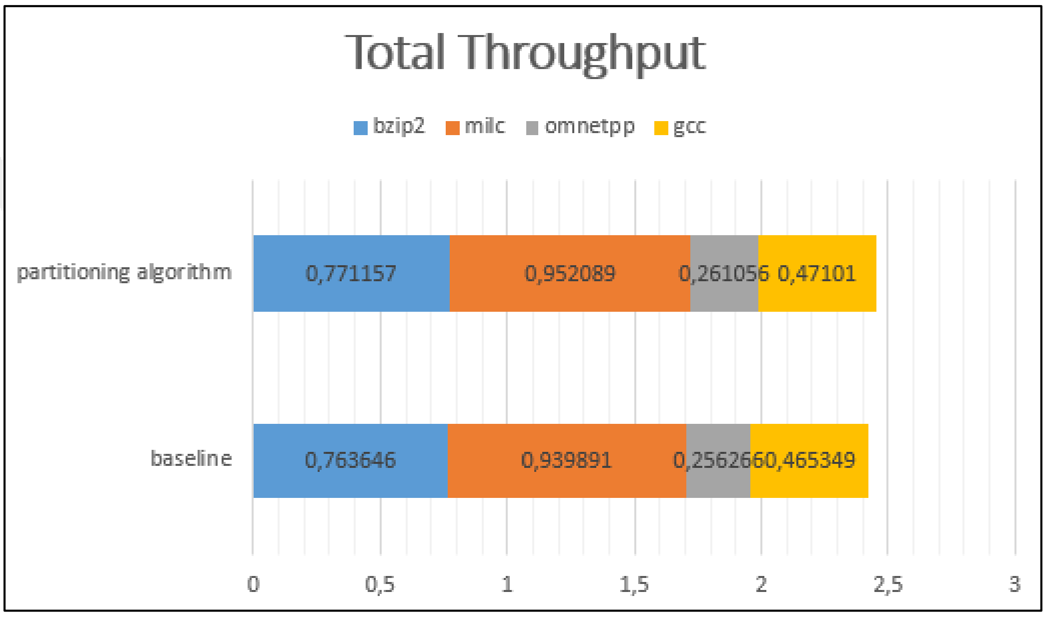


Figure 5.2. results of bzip2, milc, omnetpp and gcc

In the figure 5.2, we executed bzip2, milc, omnetpp and gcc traces concurrently. Omnetpp and gcc are considered to be harmful application by our classifier algorithm. On the other hand, bzip2 and milc are considered to be very harmful. According to the classification, our partition algorithm gave more cache resources to harmful applications than very harmful application. At the end, when we compared to baseline, the total throughput increase is approximately %2.

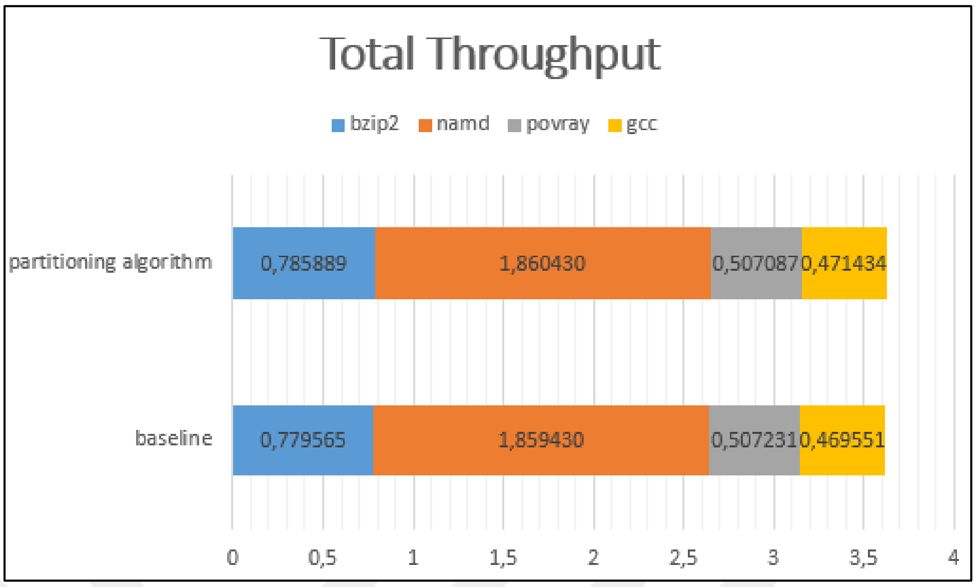


Figure 5.3. results of bzip2, namd, povray and gcc

In the figure 5.3, we executed bzip2, namd, povray and gcc traces concurrently. Bzip2, namd and povray are considered to be harmful application by our classifier algorithm. On the other hand, gcc is considered to be very harmful. According to the classification, our partition algorithm gave more cache resources to harmful applications than very harmful application. At the end, when we compared to baseline, the total throughput increase is approximately %2.

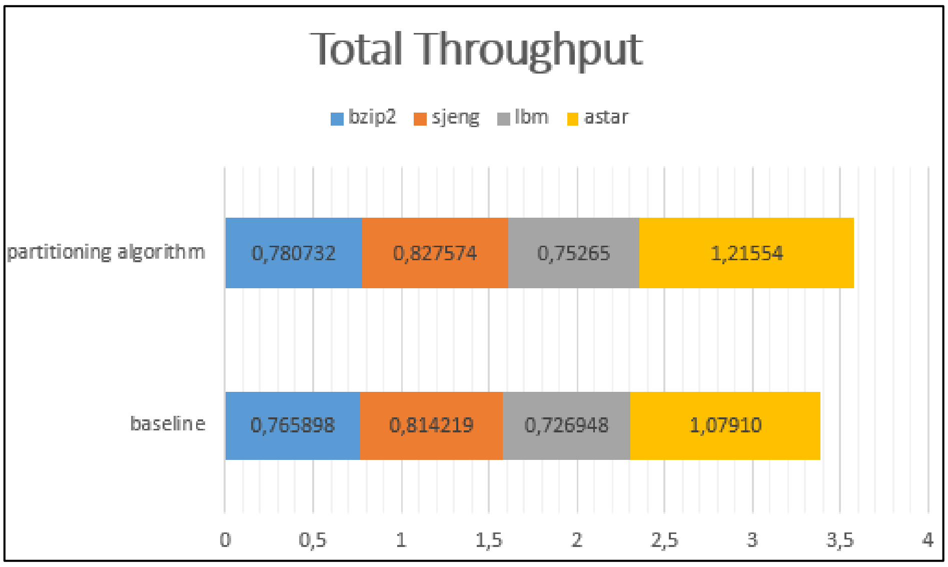


Figure 5.4. results of bzip2, sjeng, lbm and astar

In the figure 5.4, we executed bzip2, sjeng, lbm and astar traces concurrently. Bzip2 is considered to be harmful application by our classifier algorithm. Astar is considered to be harmless application by our classifier algorithm. On the other hand, sjeng and lbm are considered to be very harmful. According to the classification, our partition algorithm gave more cache resources to harmless application than harmful application. Similarly, our partition algorithm also gave more cache resources to harmful application than very harmful application. At the end, when we compared to baseline, the total throughput increase is approximately %5.

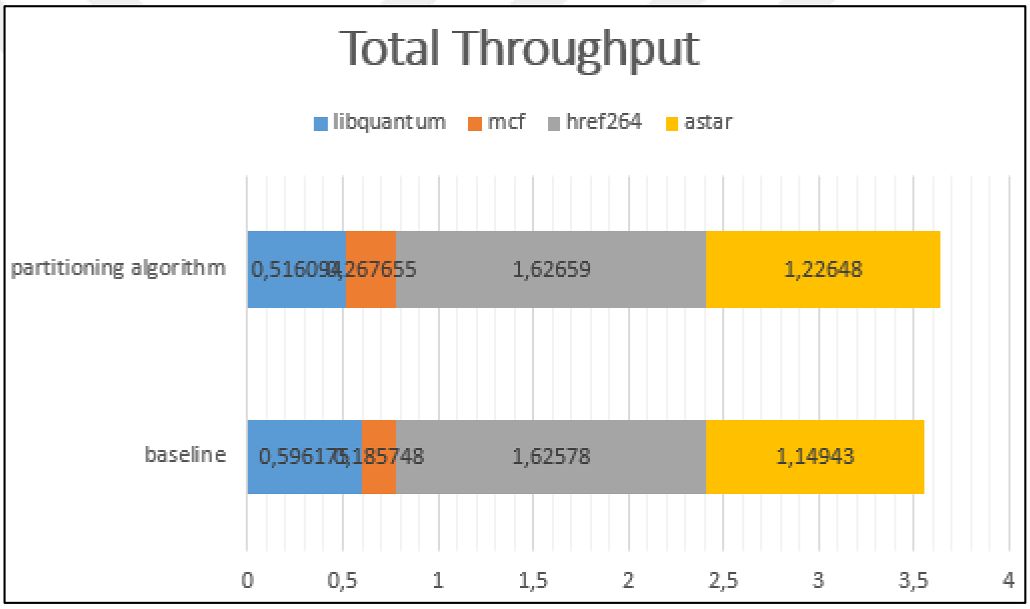


Figure 5.5. results of libquantum, mcf, href264 and astar

In the figure 5.5, we executed libquantum, mcf, href264 and astar traces concurrently. Libquantum and href264 are considered to be harmful application by our classifier algorithm. Astar is considered to be harmless application by our classifier algorithm. On the other hand, Mcf are considered to be very harmful. According to the classification, our partition algorithm gave more cache resources to harmless application than harmful application. Similarly, our partition algorithm also gave more cache resources to harmful application than very harmful application. At the end, when we compared to baseline, the total throughput increase is approximately %3.

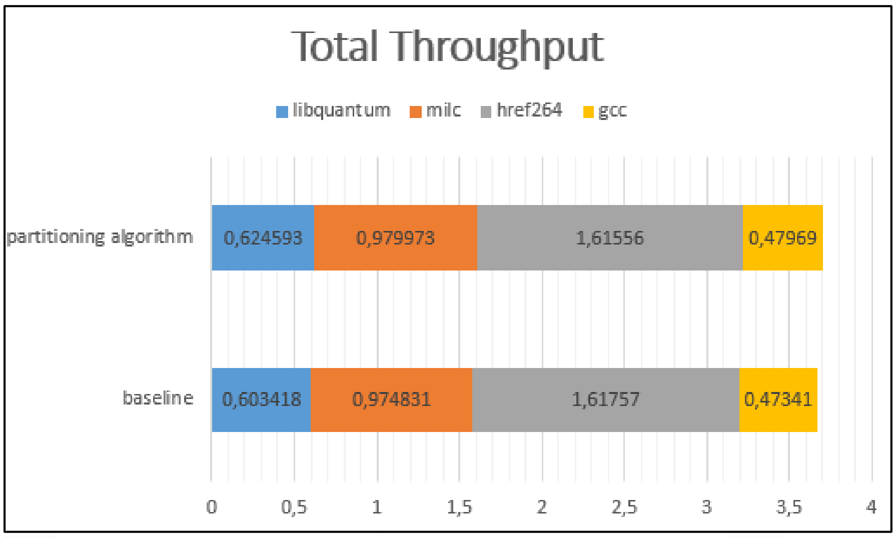


Figure 5.6. results of libquantum, milc, href264 and gcc

In the figure 5.6, we executed libquantum, milc, href264 and gcc traces concurrently. Milc and href264 are considered to be very harmful application by our classifier algorithm. Libquantum is considered to be harmless application by our classifier algorithm. On the other hand, gcc are considered to be harmful. According to the classification, our partition algorithm gave more cache resources to harmless application than harmful application. Similarly, our partition algorithm also gave more cache resources to harmful application than very harmful application. At the end, when we compared to baseline, the total throughput increase is approximately %1.

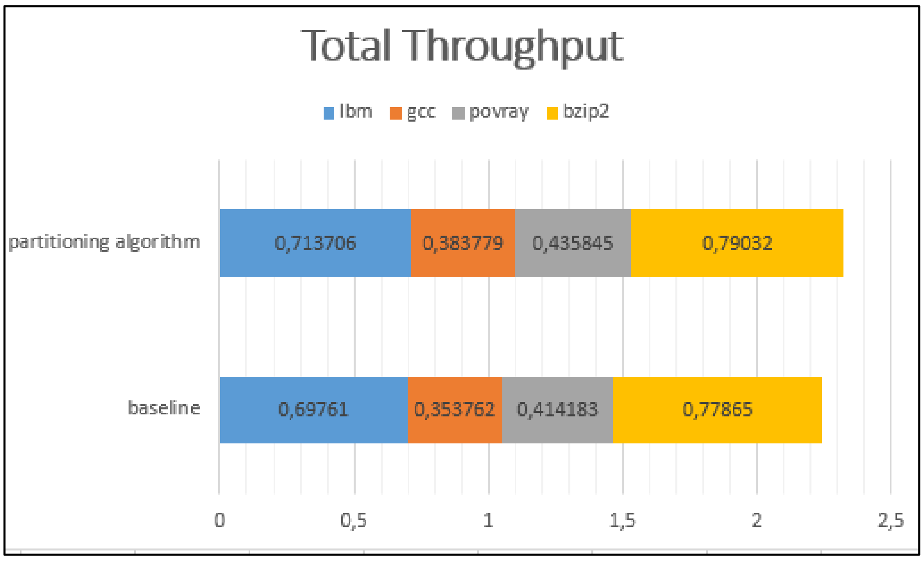


Figure 5.7. results of lbm, gcc, povray and bzip2

In the figure 5.7, we executed lbm, gcc, povray and bzip2 traces concurrently. Lbm, gcc and povray are considered to be very harmful application by our classifier algorithm. Bzip2 is considered to be harmless application by our classifier algorithm. According to the classification, our partition algorithm gave more cache resources to harmless application than very harmful application. At the end, when we compared to baseline, the total throughput increase is approximately %4.

After these quad tests, we try to evaluate fairness for eight core and eight application on the same configuration. Here are the results:

In the figure 5.8, we executed bzip2, milc, povray, gcc, omnetpp, namd, sjeng and href264 traces concurrently. Bzip2, milc, povray, gcc and namd are considered to be harmful application by our classifier algorithm. The rest of applications are considered to be very harmful. Our partition algorithm also gave more cache resources to harmful application than very harmful application. At the end, when we compared to baseline, the total throughput increase is approximately %1.

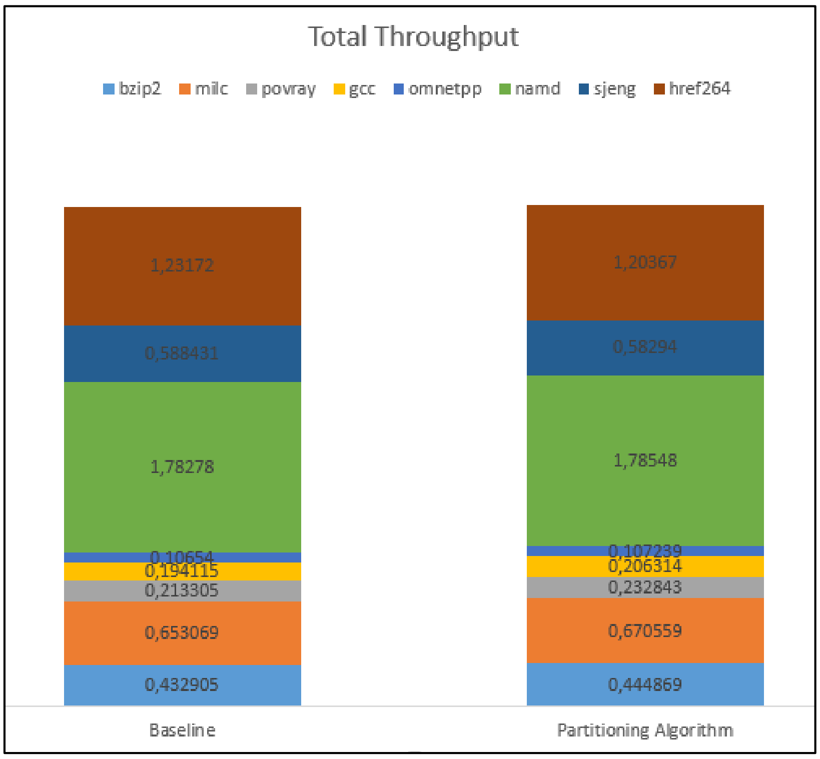


Figure 5.8. results of bzip2, milc, povray, gcc, omnetpp, namd, sjeng, href264

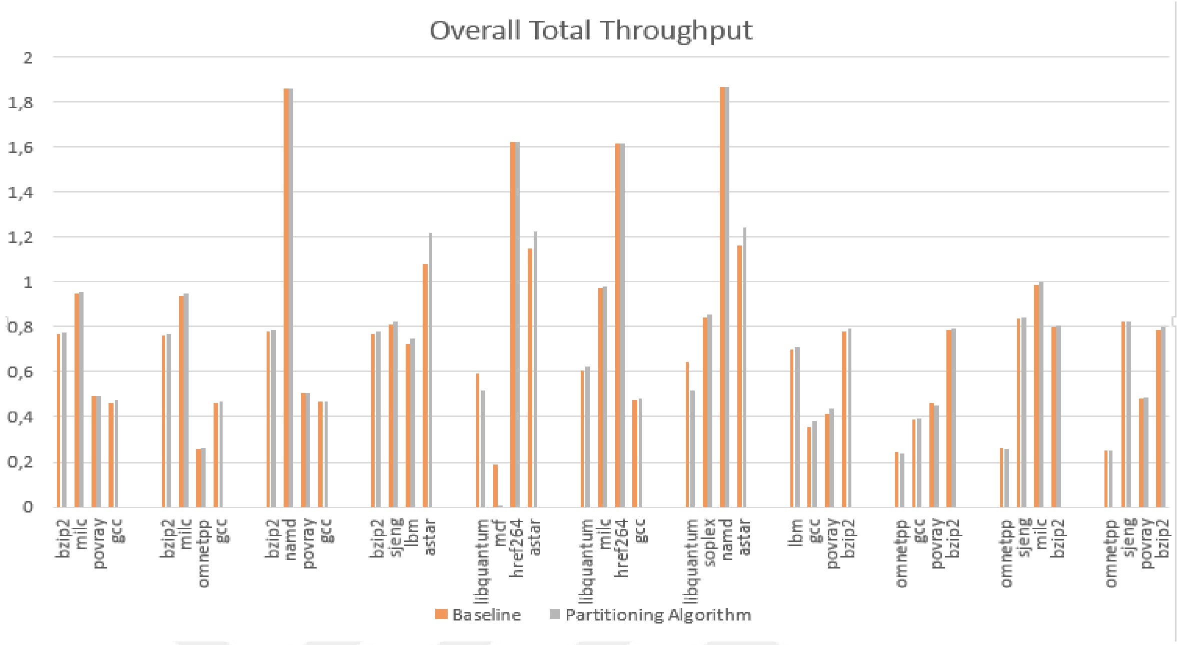


Figure 5.9. Whole Test Results

The figure 5.9. shows the whole quad tests that we done. As you see, generally the throughput value of our partitioning algorithm is bigger than baseline.

6. CONCLUSIONS AND FUTURE WORK

In this project, we tried a new cache partitioning algorithm that partitions the LLC for better performance. We designed and implemented a mechanism that classifies applications periodically at run time. Our initial method tries to give more partitions to applications that are classified as harmless. We accept these findings and the work successful enough since it provides a framework to carry on further studies in the same research area. We have four directions in our future work:

1. The classifier circuitry that we use in this study might be changed in a future study. Here, we classify an application as very harmful if it steals cache sets from other applications and does not get any benefit from accessing the LLC. However, in our tests, we find that even the applications that we classify as very harmful we observe performance degradations which we do not expect. That may mean that our classifier may need to have additional tweaking and tuning.

2. The second thing, our classification algorithm is not relative. If the range of core weights between 0 and 6, according to our proposed distribution, we obtain only "No_Operation", "Harmless", "Harmful". But we don't obtain very harmful class. In reality, the core weight value of 5 and 6, is more harmful than 3 and 4. When our classification algorithm is relative, these distribution can be like this:

Table 6.3. New distribution approach for future works

Value of core weight	**classification**
0	No_OPERATION
1,2	Harmless
3,4	Harmfull
5,6	Very Harmful

3. The third thing we can work on is the number of partitions that we assigned to different class of applications. In this study, we just tried an allocation strategy that was making the most possible sense: Very harmful applications are harmful to other applications, and, therefore, they should receive the minimum amount of cache partitions. However, other strategies need to be further investigated in a future work.

REFERENCES

1. Kunle Olukotun, Basem A. Nayfeh et al, “The Case for a Single-Chip Multiprocessor”, IEEE Transactions on Computers, 2000.

2. M. Chaudhuri et al, “Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches”, In MICRO 42, pages 401–412, New York, NY, USA, 2009. ACM.

3. Kunle Olukotun et al, “ChipMultiprocessor Architecture: Techniques to Improve Throughput and Latency”, Synthesis Lectures on Computer Architecture Book, 2007.

4. Suh, G. Edwards et al, “A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning”, in International Symposium on High-Performance Computer Architecture-8, 2002.

5. L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, “Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource”, In PACT ’06, pages 13–22, New York, NY, USA, 2006. ACM

6. S. Kim, D. Chandra, and Y. Solihin et al, “Fair caching in a chip multiprocessor architecture”, In Proc. 13th Ann. Int'l Conf. on Parallel Architectures and Compilation Techniques, pages 111-122, Sept. 2004.

7. Settle, A. et al, “Dynamically reconfigurable cache for multithreaded processors”, Journal of Embedded Computing 1(3-4), 2005.

8. J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, “Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real

systems", In International Symposium on High-Performance Computer Architecture, pages 367–378, 2008.

9. N. Rafique, W.-T. Lim, and M. Thottethodi et al, "Architectural support for operating system-driven CMP cache management", In PACT '06, pages 2–12, New York, NY, USA, 2006.

10. Qureshi, M. K. and Y. N. Patt, "Utility based cache partitioning: A low-overhead, high performance, runtime mechanism to partition shared caches", IEEE/ACM International Symposium on Microarchitecture (MICRO-39) 0-7695-2732-9/06, 2006.

11. Moreto, Cazorla, F. Ramirez and Valero et al, "Explaining dynamic cache partitioning speed ups", IEEE Computer Architecture Letters 6, 1, 1–4. M. 2007.

12. D. Sanchez and C. Kozyrakis et al, "The ZCache: Decoupling Ways and Associativity", In MICRO '43, pages 187–198, Washington, DC, USA, 2010. IEEE Computer Society.

13. D. Sanchez and C. Kozyrakis et al, "Vantage: scalable and efficient fine-grain cache partitioning", In ISCA '11, pages 57–68, New York, NY, USA, 2011. ACM.

14. S. Srikantaiah, M. Kandemir, and Q.Wang et al, "SHARP control: controlled shared cache management in chip multiprocessors", In MICRO 42, pages 517–528, New York, NY, USA, 2009. ACM.

15. K. Varadarajan, S. K. Nandy, V. Sharda, A. Bharadwaj, R. Iyer et al, "Molecular Caches: A caching structure for dynamic creation of applicationspecific heterogeneous cache regions", In MICRO 39, pages 433–442,Washington, DC, USA, 2006. IEEE Computer Society.

16. Y. Xie and G. H. Loh et al, “PIPP: promotion/insertion pseudopartitioning of multi-core shared caches”, In ISCA ’09, pages 174–183, New York, NY, USA, 2009. ACM.

17. Suh, G. Edwards et al, “Dynamic partitioning of shared cache memory”, Journal of Supercomputing, 28(1), 2004.

18. H. S. Stone et al, “Optimal partitioning of cache memory”, IEEE Transactions on Computers, 41(9), 1992.

19. D. Chiou et al, “Extending the reach of microprocessors: column and curious caching”, PhD thesis, Massachusetts Institute of Technology

20. Moreto, Cazorla, Ramirez, Valero et al, “Dynamic cache partitioning based on the MLP of cache misses. Transactions on High-performance Embedded Architectures and Compilers III, 3–23. 2011.

21. R. Iyer et al, “CQoS: a framework for enabling QoS in shared caches of CMP platforms”, In ICS-18, 2004.

22. R Manikantan et al, “Probabilistic Shared Cache Management (PriSM)”, IEEE, 2012.

23. D. Thiebaut, H. S. Stone, and J. L. Wolf et al, “Improving disk cache hit-ratios through cache partitioning”, 41(6):665-676, 1992.

24. D. Chandra, F. Guo, S. Kim, and Y. Solihin et al, “Predicting inter-thread cache contention on a chip multi-processor architecture”, In Proc. 11th Int'l Symp. on High-Performance Computer Architecture (HPCA),pages 340{351, Feb. 2005.

25. *Standard Performance Evaluation Corporation*, SPEC, http://www.spec.org/benchmarks.html, 1995-2011.

26. A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum et al, “Performance of multithreaded chip multiprocessors and implications for operating system design.” In Proc. 2005 USENIX Technical Conference, pages 395-398, 2005.