SET-BASED DYNAMIC CACHE PARTITIONING ON CHIP MULTIPROCESSORS

by

Nazlı Tokatlı

Submitted to Graduate School of Natural and Applied Sciences
in Partial Fulfillment of the Requirements
for the Degree of Master of Science in
Computer Engineering

Yeditepe University
2016

SET-BASED DYNAMIC CACHE PARTITIONING ON CHIP MULTIPROCESSORS

APPROVED BY:

Assist. Prof. Gürhan Küçük (Thesis Supervisor)	....
Assist. Prof. Esin Onbaşıoğlu	....
Dr. Yaşar Safkan	....

DATE OF APPROVAL: ..../..../2016

ACKNOWLEDGEMENTS

I would like to record my gratitude to Assiss Prof. Gürhan Küçük for his supervision, advice, and guidance from the very early stage of this research as well as giving me extraordinary experiences throughout the work. Above all and the most needed, he provided me unflinching encouragement and support in various ways. His truly scientist intuition has made him as a constant oasis of ideas and passions in science, which exceptionally inspire and enrich my growth as a student and a researcher. I am indebted to him more than he knows.

It is a pleasure to thank all my friends in the computer engineering department laboratories and administration staff of Yeditepe University Mine Mesta, Bilgin Koşucu, Kerem Irgan, Ahu Soydan, Murat Birben and Suna Dündar for their help and support to finish this project.

My parents Ertam Nakip and Güler Nakip, who have always supported, encouraged and believed in me and who so lovingly and unselfishly cared for my kids.

Special thanks go to my best partner and friend, my husband. Ali, you gave me your unconditional support and love through all this long process.

ABSTRACT

SET-BASED DYNAMIC CACHE PARTITIONING ON CHIP MULTIPROCESSORS

Today, most of the chip multiprocessor architectures utilize a shared second level cache to reduce the off-chip memory delay. However, benefit from such a cache may be very limited due to cache conflicts caused by applications running in parallel. The alternative approach of having a private second level cache dedicated to each core is also problematic, since there are always applications with large memory footprints or shared address space requirements. In the literature, there are numerous studies that try to partition the second level cache. These studies generally focus on dedicating an appropriate number of ways and policies to each core according to the runtime memory requirements of applications. On the contrast, this study proposes a mechanism to dynamically partition the cache based on sets. In this mechanism, the resizing decisions for each logical partition are made according to the runtime statistics collected by the hardware at periodic time intervals. Since the mechanism focuses on cache sets rather than cache ways, the resizing of the cache partitions can be done in a finer-grain, any cache policies can be freely chosen and, the additional complexity requirements can be kept at minimum compared to other schemes. When compared to the shared baseline cache configuration, the performance (throughput) gain in workloads containing solely memory-intensive applications is as much as 9%, on the average. For hybrid workloads that run memory- and computation-intensive applications together, the performance is improved by more than 15% on the average across all simulated application mixtures.

ÖZET

SET-BASED DYNAMIC CACHE PARTITIONING ON CHIP MULTIPROCESSORS

Günümüzde çoklu çip işlemci mimarileri, çip dışındaki bellek gecikmelerini azaltabilmek için paylaşımlı bir ikinci seviye önbellek kullanmaktadırlar. Ancak bu tip bir önbellekten elde edilecek fayda, birlikte çalıştırılacak uygulamalardan kaynaklanan önbellek sorunları sebebiyle çok kısıtlı olabilir. Her çekirdeğe yönelik özel bir ikinci derece önbellek kullanma konusundaki alternatif yaklaşım da sorunludur. Çünkü aşırı bellek gereksinimi olan veya paylaşımlı adres alanı ihtiyacı olan uygulamalar her zaman mevcuttur. Literatürde ikinci seviye önbelleği bölümlere ayırmaya çalışan çok sayıda çalışma vardır. Bu çalışmalar genellikle, uygulamaların çalışma-anı bellek gereksinimlerine göre her bir çekirdeğe uygun sayıda önbellek öbeklerinin ve denetimlerinin atanmasına odaklanmaktadırlar. Bu çalışma ise diğer çalışmaların aksine önbelleğin küme tabanlı olarak dinamik ölçeklenmesine dayalı bir mekanizma önermektedir. Bu mekanizmada, her mantıksal bölüm için gerekli boyutlandırma kararları, düzenli aralıklarla donanım tarafından toplanan çalışma-anı istatistiklerine göre alınır. Mekanizma, önbellek öbeklerinden ziyade kümelerine odaklandığından, önbellek bölümlerinin yeniden boyutlandırılması daha ince ayrıntılı olarak yapılabilir; herhangi bir önbellek politikası serbestçe seçilebilir ve ilave karmaşıklık gereksinimleri diğer metotlara kıyasla daha düşük seviyede tutulabilir. Sonuçlar, paylaşımlı taban seviyesindeki önbellek yapılandırması ile karşılaştırıldığında, sadece yoğun bellek kullanan uygulamaları içeren iş yükündeki performans (toplam iş çıktısı) artışı, ortalamada %9'a kadar çıkmaktadır. Belleği ve hesaplama yoğunluklu uygulamaları bir arada çalıştıran melez iş yükleri için, performans, tüm benzetilmiş uygulama kombinasyonları göz önüne alındığında %15 artmaktadır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS / ABBREVIATIONS

D_i	SBPMDTA queue elements
n	The number of elements in the SBPMDTA queue
$\sum$	Sigma/summation
ALU	Arithmetic Logic Unit
ATD	Auxiliary Tag Directory
B/W	Band Width
CMP	Chip Multiprocessors
Dcache	Data Cache
FATL	Fixed Address Translation Logic
HSHR	Hit Status Holding Register
HL2UC	Hybrid Level Two Unified Cache
ILP	Instruction Level Parallelism
IPC	Instruction Per Cycle
Icache	Instruction Cache
IQ	Issue Queue

L2	Level Two
LRU	Least Recently Used
LSQ	Load/Store Queue
MRU	Most Recently Used
MSHR	Miss Status Holding Register
MLP	Memory Level Parallelism
O/S	Operating System
PL1C	Private Level One Cache
PL2C	Private Level Two Cache
ROB	Reorder Buffer
RF	Register File
SL2UC	Shared Level Two Unified Cache
SRAM	Static Random Access Memory
SBP	Set Based Partitioning
SBPATL	SBP Address Translation Logic
SBPMDTA	SBP Miss Rate Difference Threshold Average

SBPWT	SBP Write Through
TLP	Thread Level Parallelism
UMON	Utility Monitor
WIPC	Weighted IPC

1. INTRODUCTION

The main aim on utilizing multiple processor cores on a single packaging is to improve the throughput of a machine by running multiple applications in parallel. Although, this seems a very natural approach to improve the performance of a machine, until recently people were opting to increase the performance of single processor architectures rather than following the multiprocessor direction. In recent years, with the recent progress in the process and memory technologies, machine architectures with multiple processor cores have become more and more feasible, and started to appear in the electronic markets.

Today, a typical multiprocessor architecture supplies dedicated private caches to each core in the first level, and most of the temporal locality in the data streams is captured in that same level. However, the first level caches are designed and implemented with performance concerns, and, as a result, the size of these caches is usually not large enough to exploit the spatial locality in the data streams. For capturing the spatial locality, a larger second level cache is utilized, and this cache level is usually shared by all the processor cores like in Sun Niagara, IBM Power5, and Intel Core architectures. Such a shared cache level is ideal especially for applications that are in producer/consumer relation (shared workloads).

However, when the applications, running on different processor cores, are not dependent on each other and are executing privately (private workloads), a shared cache level may indeed incur performance penalties. A typical example may be given with two private workloads accessing the shared level cache and stealing cache lines from each other. In such cases, the hit rate to the second level cache for each application will be much lower than the hit rate when each application experiences when they run standalone.

In this thesis, we propose a mechanism that logically partitions the second level shared cache in multiprocessor architectures. During execution, our partitioning algorithm changes the number of sets dedicated to each core according to the memory requirements of each application running on those cores. As a result, each core can get its own private partition in the second level cache. For determining the memory requirements of each application, and resizing the partitions, we propose a periodic sampling mechanism that collects the memory statistics of each core, and a decision logic that tries to partition the cache according to those statistics in order to improve the throughput of the machine.

1.1. BACKGROUND

1.1.1. Chip Multiprocessors

Chip multiprocessors (CMP) are simply single thread processor cores with private and/or shared memory systems. Each core exploit moderate amount of instruction level parallelism within any one thread (ILP), and in the same time, it executes multiple threads in parallel across multiple processor cores (TLP).

CMP architectures are preferable over superscalar architectures in many terms, such as performance, complexity, power and reliability. CMP increase the total throughput of running applications without need to increase IPC using aggressive mechanisms such as speculative execution and out of order execution. Besides, CMP architectures are decentralized allowing load balancing among running applications, where each application has its own resources to use and eventually prevent the increase in the die temperature. When the reliability is the concern, decreasing hot spots on cores will increase the life time of the chip.

In terms of complexity, CMP architectures are less complex compared to superscalar architectures. Since CMP do not need any additional hardware mechanisms to extract parallelism, Instead of additional hardware we can increase cache size for each core or insert two or three additional cores inside silicon space.

Figure 1.1 shows chip multiprocessors architecture which cosists eight cores on a single die area. Each core contains its own issue logic, architectural register and physical registers, branch predictor table and level one data and instruction caches.

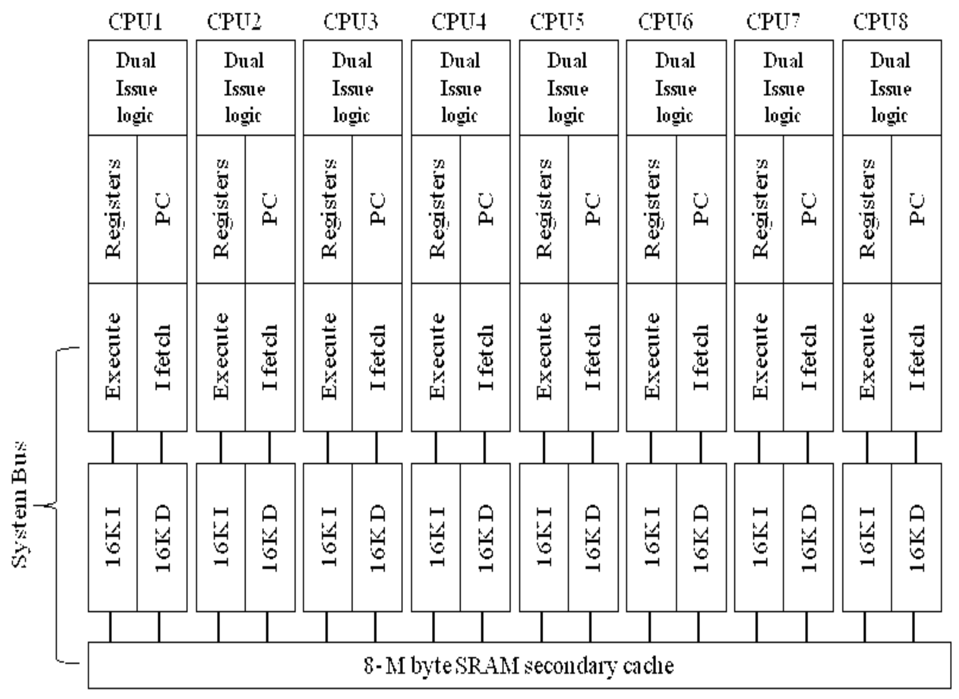

Figure 1.1. Chip multiprocessor architecture

There are two categories of CMPs: homogenous and heterogeneous. First type involves geometrically increasing the number of cores with each advance in feature size. This means either duplicating, or quadrupling a same core, and interconnecting them to produce a more powerful core. Second type has both a high and low complexity cores where applications are mapped to cores in such a way that each application executes on a core that best fits its resource requirements. Figure 1.2 shows examples for both types.

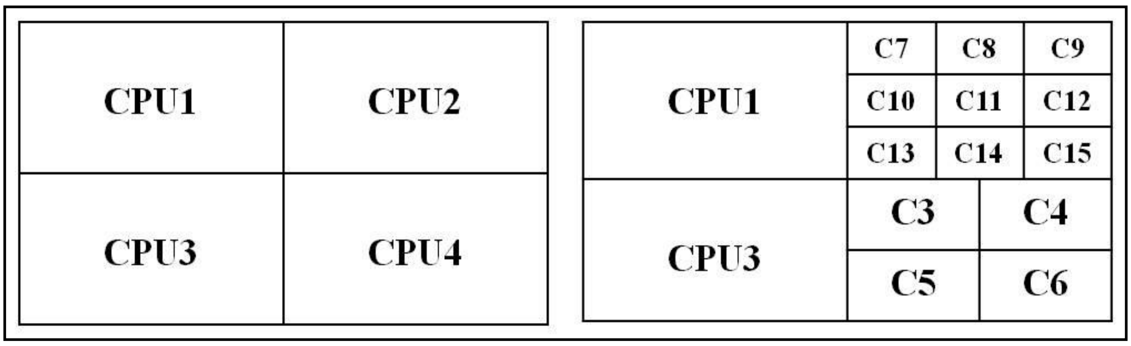

Figure 1.2. Symmetric and asymmetric CMP in the same die area

1.2. MEMORY ORGANIZATION

In CMP architectures various memory configurations are proposed but the main configurations that used nowadays are as follows:

- First level data and instruction caches, second level private cache for each core. Figure 1.3 shows the first configuration.
- First level data and instruction caches, second level shared cache among cores. Figure 1.4 shows the second configuration.

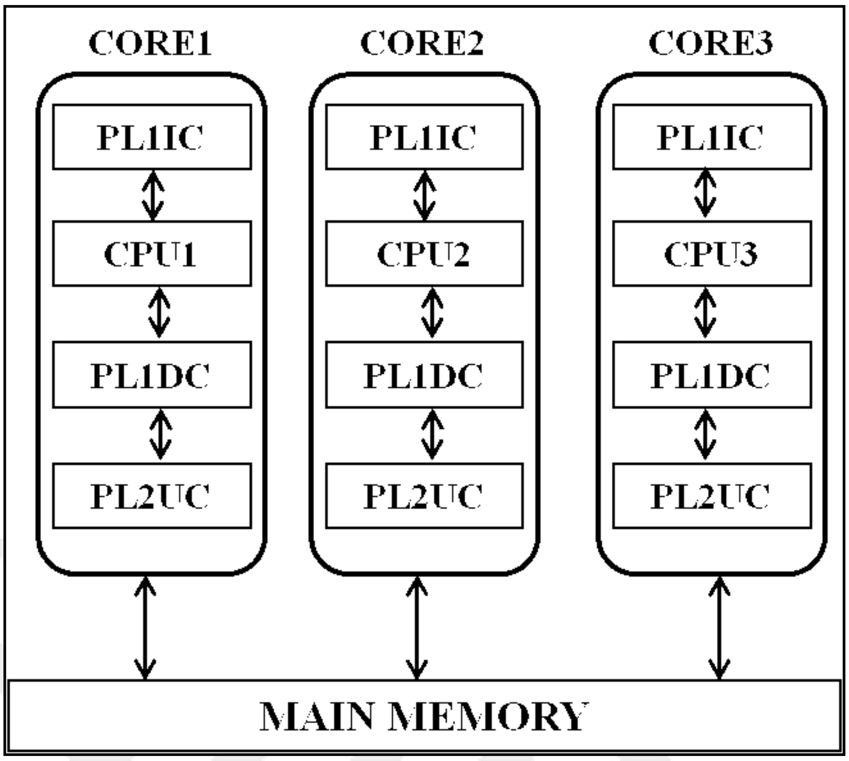

Figure 1.3. PL1C and PL2C Configuration in CMP

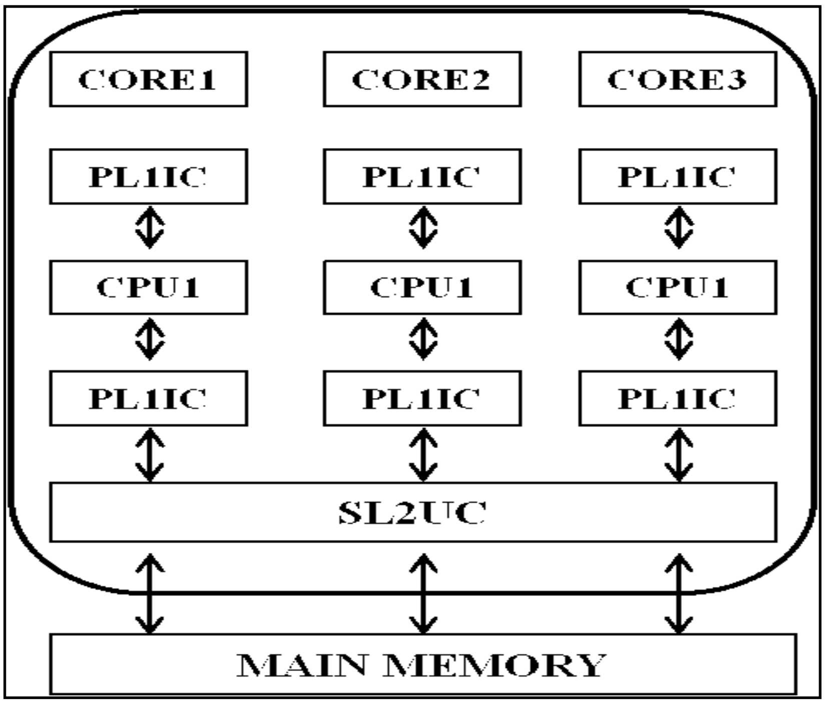

Figure 1.4. PL1C and SL2C Configuration in CMP

1.2.1. Level two private cache

One of the cache configurations that are commonly used in CMP architectures is with a second level private cache space dedicated to each core. The main reason behind using such configuration is that the data in the cache space resides close to the core reducing the access latency. Moreover, reducing the distance between the data and processor has major impact in reducing power dissipation. On the other hand the private level two cache not very flexible configuration cause a core cannot use the cache space of idle core(s).

Second level private cache configuration is implemented in AMD’s Athlon™ 64 X2 Dual-Core processors. Furthermore, the private level two caches gives maximum performance gains when the running workload types are private or when there is data sharing between them.

1.2.2. Level two shared cache

Shared second level cache is another widely used configuration in CMP architectures, since it provides flexibility. Furthermore, a copy of data can be accessed by all cores resulting in decreasing off-chip access. However, shared level two caches have certain disadvantages such as data pollution since they can be accessed by all cores. Moreover, when the distance between the cache and the processor is not close, higher hit latencies are expected.

Furthermore additional communication networks necessary between level one cache and second level cache and extra directory bits for keeping track of data owners in the other words cache coherency protocols are needed.

Lastly the shared level two cache cause unfairness among running threads, especially if one of the running application is memory intensive which always access the cache and continuously invalidate the computational application data.

Examples of second level shared cache configuration is implemented in Sun Niagara, IBM Power5, and Intel Core architectures. Furthermore, the shared level two caches give maximum performance gains when the running workload types are in producer/consumer relation.

1.3. MOTIVATION

Introduction of cache levels to each core in CMP architectures adds new requirements to the hardware, such as additional circuitry necessary for running a cache coherency protocol, additional die area for the SRAM structure that stores the cache levels, additional wiring to move data to/from the cache, etc. Although, they require such design and implementation complexities, caches are valuable structures for reducing the performance penalties caused by the memory bottleneck of the Von Neumann architecture.

On-chip second level cache configurations as private or shared do not maximize the performance of CMPs. We believe the workload type is an important factor in determining the best second level cache configuration that later leads to reduce the off-chip accesses and increase in CMP performance. Moreover, there is a need for alleviating cache unfairness among threads by defining a run time mechanism that decide the amount of cache space required by each core.

The benefits we get from our proposed design in this thesis are two folds. First, we provide a private and an adaptive partitioning mechanism for each core, and we reduce the performance penalties due to cache steals resulting from running private workloads [1].

Second, we still keep an adaptive shared partition inside this cache level, and thus, we can improve the performance of the machine when the processor cores are running shared workloads. Additionally, this shared partition may act as a buffer area for private workloads to prevent them stealing cache resources from each other.

Finally, there are many cache-way partitioning studies in the literature. However, to the best of our knowledge, this work is one of the first studies that propose *logical* cache-set partitioning to improve the processor performance. Here, we would like to further emphasize the *logical* keyword: in this study, we do not change the physical organization of the cache, but only change the procedure for accessing the cache sets. To achieve this, we propose an additional circuitry which we call the Set-Based Partitioning (SBP) Address Translation Logic. The details of the proposed algorithm and logic are explained in Chapter three.

There are many advantages of a set-based cache partitioning mechanism compared to a way-based partitioning mechanism. Here, we list some of them to motivate our study:

- Finer-grain of control on a typical second level cache. There are much more cache sets than cache ways. When the caches are partitioned based on ways, the minimum resizing amount can be set in a much coarser-grain, since we dedicate at least set number of additional cache blocks to an application. 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 the organization can no longer be used. On the contrast, when a set-based partitioning is utilized, no modifications are required on the 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, on the other hand, even a fully-associative cache configuration might be feasible, and the number of counters necessary to collect statistics is limited by the number of cores.

1.4. THESIS OUTLINE

The thesis organization as follow: In Chapter two, the background study about different methods that used in partitioning second level cache among running applications based on variety range of matrices. In Chapter three we give a detailed explanation of our design proposal for partitioning second level cache followed by our experimental methodology in Chapter four. In Chapter five we present our simulation results and include our discussions. Finally, in Chapter six we conclude our study by summarizing the obtained results and providing additional ideas that can be implemented as a future work**.**

2. RELATED WORK

Caches are crucial structures that enable us to fight with the well-known *Memory Wall* problem. SMP and CMP are new architectures, which they offer the opportunity to obtain higher throughputs by allowing TLP, face the challenge of sharing resources such as last levels of shared caches. Consequently, there is a need for efficient usage of last level caches to prevent any starvations among running threads.

There are numerous studies that propose new strategies and configurations related to shared caches. In this section, we give background information specifically on adaptive caches.

2.1. STATIC L2 CACHE PARTITIONING

Stone et al [2] examined static partitioning of the cache memory among processors. They believe that LRU which explicitly partition the cache based on demands of applications is not the optimal policy. They proposed using a referenced stream miss rate as a function of cache allocation size of individual competing processes. Chiou et al [3] propose static partition of L2 caches by giving specific number of cache ways to running threads based on profiling information for each thread. The main disadvantage of static cache partitioning is that it is not very flexible. Assigning fixed partitions from a cache to applications is not a very good idea, especially if one of the running applications is memory-intensive and the other is computational-intensive. In that case, inefficient resource usage of caches is imminent.

2.2. DYNAMIC L2 CACHE PARTITIONING

The dynamic partitioning of shared caches is firstly investigated by Suh et al [4] [5]. 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. This gain which is proportional to the cache miss rate as a function of cache size.

The overall flow of the partitioning scheme can be viewed as a set of four modules: on-line cache monitor, O/S processor scheduler, partition module, and cache replacement unit. The scheduler provides the partition module with the set of executing processes that shares the cache at the same time. Then, the partition module uses this scheduling information and the marginal gain information from the on-line cache monitor to decide a cache partition. The proposed module uses a greedy algorithm to allocate each cache block to a process that obtains the maximum marginal gain by having one additional block. Finally, the replacement unit maps these partitions to the appropriate parts of the cache.

Kim et al introduce fairness metrics for only a shared resource which is the L2 cache, in CMP architectures [6]. These metrics keeps O/S scheduler from three main problems which are:

- Thread starvation, which happens when one thread fails in competing for sufficient cache space necessary to make satisfactory forward progress.
- Priority inversion, where a higher priority thread achieves a slower forward progress than a lower priority thread, despite the attempt by the O/S to provide more time slices to the higher priority thread. This happens when the higher priority thread loses to the lower priority thread (or other threads) in competing for cache space.
- Forward progress rate of a thread is highly dependent on the thread mix in a co-schedule[1]. This makes the forward progress rate difficult to characterize or predict, making the system behaviour unpredictable.

  The fairness metrics that are used in partitioning (based on ways) the cache statically and dynamically are represented in equation (2.1), (2.2), (2.3), (2.4), (2.5) respectively.

---

[1]Co-schedule: assignment of threads with different characteristics by operating system to available CPUs, as example of such mix a memory-intensive thread with a computational-intensive thread.

$$M_1^{ij} = |X_i - X_j|, where\ X_i = \frac{Miss\_shr_i}{Miss\_ded_i} \tag{2.1}$$

$$M_2^{ij} = |X_i - X_j|, where\ X_i = Miss\_shr_i \tag{2.2}$$

$$M_3^{ij} = |X_i - X_j|, where\ X_i = \frac{Missr\_shr_i}{Missr\_ded_i} \tag{2.3}$$

$$M_4^{ij} = |X_i - X_j|, where\ X_i = Missr\_shr_i \tag{2.4}$$

$$M_5^{ij} = |X_i - X_j|, where\ X_i = Missr\_shr_i - Missr\_ded_i \tag{2.5}$$

- $Miss\_shr_i$ It represents the number of misses for thread i, when it is sharing the cache with other running threads.
- $Missr\_shr_i$ The Miss rate for thread i, when it is sharing the cache with other running threads.
- $Miss\_ded_i$ The Number of cache misses for thread i, when it runs alone in the system.
- $Missr\_ded_i$ Cache miss rate for thread i, when it runs alone in the system.

Hsu et al 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 which is the case for a utilitarian policy. They propose the usage of instruction per cycle and misses per access matrices to decide the allocation process of cache resource among competing applications in CMPs. Additionally, they use weighted IPC (WIPC) metric, which is equal to IPC of thread when it is run alone divided by the IPC for a thread running with other competing threads in the system. Then, they apply both the utilitarian model (maximizing WIPC, i.e., minimizing aggregate relative degradation) and the communist model (equalizing WIPC, i.e., equalizing relative degradation across all threads). Finally, they conclude that using a traditional cache replacement policy such as LRU and performing static cache partitioning are not sufficient to provide near optimal performance. They state that some thread-aware cache resource allocation mechanism is required, and use of communist or utilitarian policy for partitioning cache in CMP may not work perfectly for some type of workloads [7].

Settle et al also investigate a dynamic cache partitioning mechanism, again 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 the chances of process utilization [8]. They need to modify the LRU policy to collect the reuse information of the running threads in the system. When the thread id of the 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 that is used in least frequently used cache replacement policies. If the reuse rank of the candidate thread relative to the other cache lines in the set is higher than a threshold value, the line is not considered for eviction. Instead, the LRU - 1 line is evaluated and the process repeats itself. If there are no options available under this scheme, the algorithm reverts back to the normal LRU candidate. By consulting the reuse information of each potential victim cache line, this algorithm helps increase the time that data from another thread stays in the cache. Thus, in the 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.

Lin et al propose partitioning the cache based on an O/S technique called *page coloring.* A page color is 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 for running programs. Limiting the physical memory pages within a subset of colors enables the O/S to limit the cache used by a given process to cache regions of those colors. On the other hand, when a decision is made to increase the cache resource of a given process, i.e. increasing the number of colors used by the process, the kernel will enforce the decision by re-arranging the virtual-physical memory mapping of the process. If the number of colors used is m, then all virtual memory pages, if not paged out, are mapped onto physical memory pages of those colors. When the number of colors increases to m+1, the kernel assigns one more color to the process and move roughly one of every m+1 of the existing pages to the new color. This process involves allocating physical pages of the new color, copying the memory contents and freeing the old pages. When a decision is made to reduce cache resources, the kernel will also recolor a fraction of virtual memory pages, accordingly. Moreover a hardware mechanism may support a finer granularity of cache allocation when it is needed. Finally, a sufficient software approach used to achieve cache partitioning in operating systems through memory address mapping. The authors claim that the proposed software approach can further be used as a tool to evaluate the hardware design and performance in multi-core architectures [9].

Rafique et al propose using a hardware quota enforcement mechanism to manage shared caches in CMP while a communication between the hardware and O/S establish to apply a wide variety of policies by tuning the quotas during regularly scheduled O/S interventions [10]. Disadvantage of this work is the limitations of the proposed hardware mechanism that only supports a coarse granularity of cache allocation.

Qureshi et al partition the ways in the cache 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 [11]. The process of information collection by UMON to decide cache partitioning is based on stack distance profiling. Each set in a cache can be seen as a LRU stack, where lines are sorted by their last access cycle. In that way, the first line of the LRU stack is the most recently used (MRU) line while the last line is the LRU line. For a *k*-way associative cache with a LRU replacement algorithm, there is a need for *k*+1 counters: *C*1, *C2*, . . . *Ck*, *C>k*. On each cache access, one of the counters is incremented. If it is a cache access to a line in the *ith* position in the LRU stack of the set, *Ci* is incremented. If it is a cache miss, the line is not found in the LRU stack and, as a result, we increment the miss counter *C>k*. Stack distance profile characteristic is that the number of cache misses for a smaller cache with the same number of sets can be easily computed using the stack distance profile.

For example, for a *k'*-way associative cache, where ***k' < k***, the new number of misses can be computed as:

$$Misses = C{>}k + \sum_{i=k'+1}^{k} c_i \tag{2.6}$$

Using the stack distance histogram of two applications, the UMON can derive the optimal L2 cache partition that would minimize the total number of misses, as this last number corresponds to the sum of misses of each thread with the assigned number of ways.

Moreto et al propose dynamic cache partitioning to maximize the total throughput of running threads by minimizing the total cost. A partitioning algorithm assigns higher cost to isolated L2 misses due to its higher impact on performance and giving lower cost to clustered L2 misses. The cost assigning process implemented by extra hardware, which are auxiliary tag directory (ATD), miss status holding register (MSHR) and hit status holding register (HSHR).

The job of ATD is keeping track of the L2 accesses for any possible cache configuration. Independently of the number of ways assigned to each core, storing the tags and LRU counters of the last K accesses of the thread, where K is the L2 associativity. The Miss Status Holding Register (MSHR) and the Hit Status Holding Register (HSHR) are used to compute the MLP cost of the access. The MSHR and HSHR are similar to an L2 miss buffer and are used to hold information about any load that has missed or hit in the L2 cache. The modified L2 MSHR and HSHR have one extra field that contains the MLP cost of a miss or a hit. Moreover it also stores the 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. Their proposed design performs 10 per cent better over traditional eviction policies with LRU [12] [13].

3. SET BASED PARTITIONING CACHE

This chapter explains the design and implementation details of our proposed design. We introduce two additional counters that keep tracking cache accesses and cache misses. Additionally, we introduce a hardware control mechanism that runs our partitioning algorithm. As shown in Figure 3.1 the partitioning algorithm decides and changes the size of each partition on shared second level cache, dynamically. Finally there is no need for modifying the cache replacement algorithm, since the partitions are decided at set granularity. The details of the partition algorithm are given in the following section.

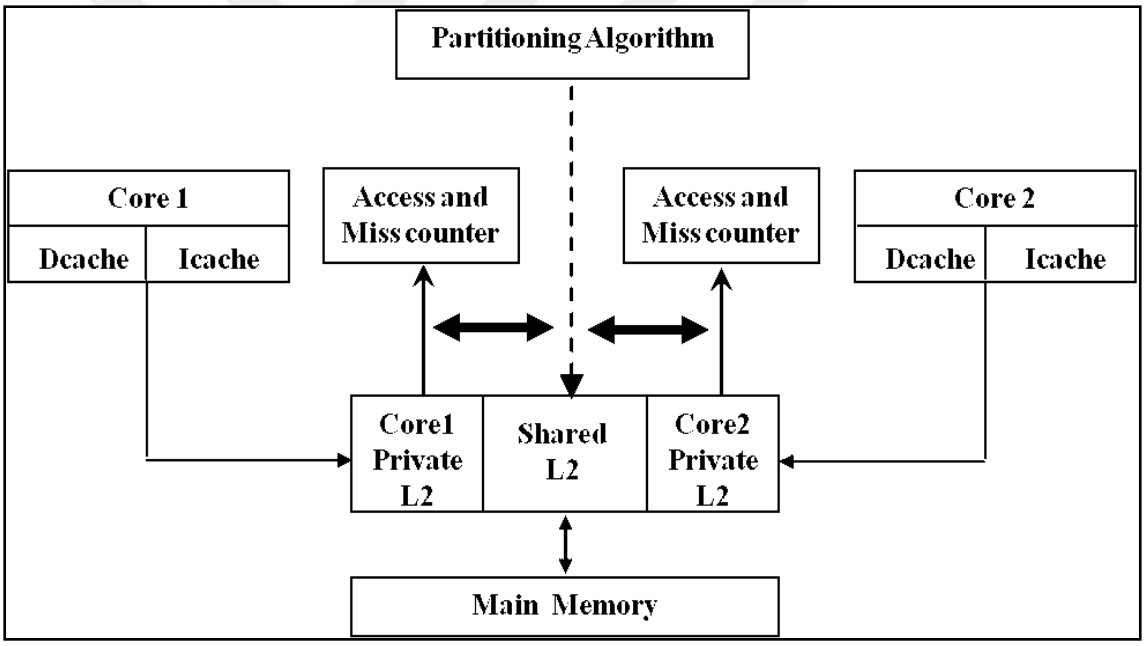

Figure 3.1. SBP cache

3.1. PARTITIONING ALGORITHM

The main aim of our partitioning proposal is to decrease the number of misses or (miss rate) for each core when accessing second level cache to achieve better performance when compared to baseline SL2UC. The partitioning algorithm flowchart is given in Figure 3.2. Note that after initializing logically private and shared partitions on the second level cache the partition algorithm goes in to an infinite loop. Inside this loop, we collect cache access and miss statistics for each core until an *update interval* ends. These statistics are collected in dedicated hardware counters. At the end of each *update interval*, the dynamic miss rate for each core is calculated as follows:

Dynamic miss rate =

Number of miss (interval) /Number of total access (interval)

(3.1)

Here, the dynamic miss rate is the miss rate for that specific *update interval*. As a result it indicates the instant value of the miss rate rather than its cumulative value, and it is a useful parameter for tracking down the memory requirements of the running applications. The final step in the algorithm is the calculation of the dynamic miss rate difference between cores. This parameter is calculated as shown in equation (3.2).

Dynamic miss rate difference =

|dynamic miss rate core0 - dynamic miss rate core1|

(3.2)

When the dynamic miss rate difference value is less than a specific *SBPThreshold* value**,** we keep the existing configuration; otherwise, we increase the partition size of the core with the higher miss rate by *s* sets. Additionally, we update the offset and the size of private partition as well as the size of the shared partition. This update process is realized by the SBP cache address translation logic.

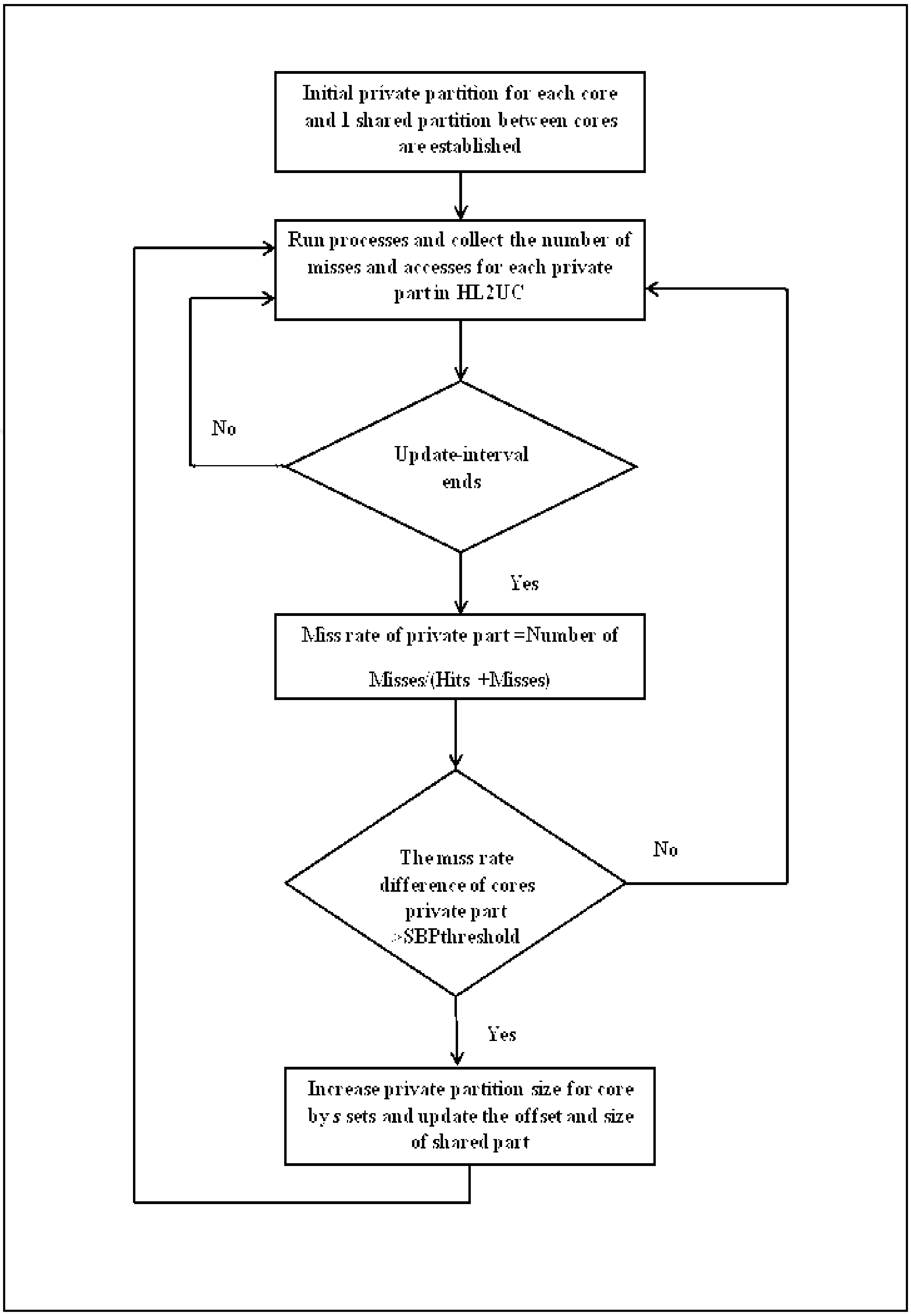
Initial private partition for each core and 1 shared partition between cores are established
Run processes and collect the number of misses and accesses for each private part in HL2UC
Update-interval ends
No
Yes
Miss rate of private part =Number of Misses/(Hits +Misses)
The miss rate difference of cores private part >SBPthreshold
No
Yes
Increase private partition size for core by s sets and update the offset and size of shared part

Figure 3.2. Set based partitioning algorithm flowchart

3.1.1. SBP cache address translation logic (SBPATL)

The SBP address translation logic is used to translate a physical address to the appropriate partition. In Figure 3.3, the logical layout of the SBP cache and its address translation logic are shown. For instance, when the *Core one private* partition is being accessed, any physical address is mapped to cache sets between *m* and *x* by the SBPATL.

Figure 3.4 shows a detailed example to further explain this mechanism. A physical address is divided into three fields as tag, set and block offset. In a typical cache access the extracted set field is used to locate the cache set where the data is assumed to be located. In that case, the raw decoder selects the cache set and then the tag bits of the physical address and the tag bits of the all cache ways are compared. When any of these tag fields of the cache ways matches with the tag of the physical address a cache hit occurs and the corresponding data block on the cache way is accessed. In a SBP cache, an additional control circuitry, which we call SBPATL, is located between the address bus that supplies the physical address and the cache raw decoder. In this example, the extracted set value is 800. This set number and the accessing core number (i.e. one, for this example) are supplied to the SBPATL, which knows that core number one is using cache sets between 1000 and 1500. The equation (3.3) given below is used to compute the new accessed set number (1300, for this example).

Accessed Set=
(Original set from physical address% core's private partition size)+core offset

(3.3)

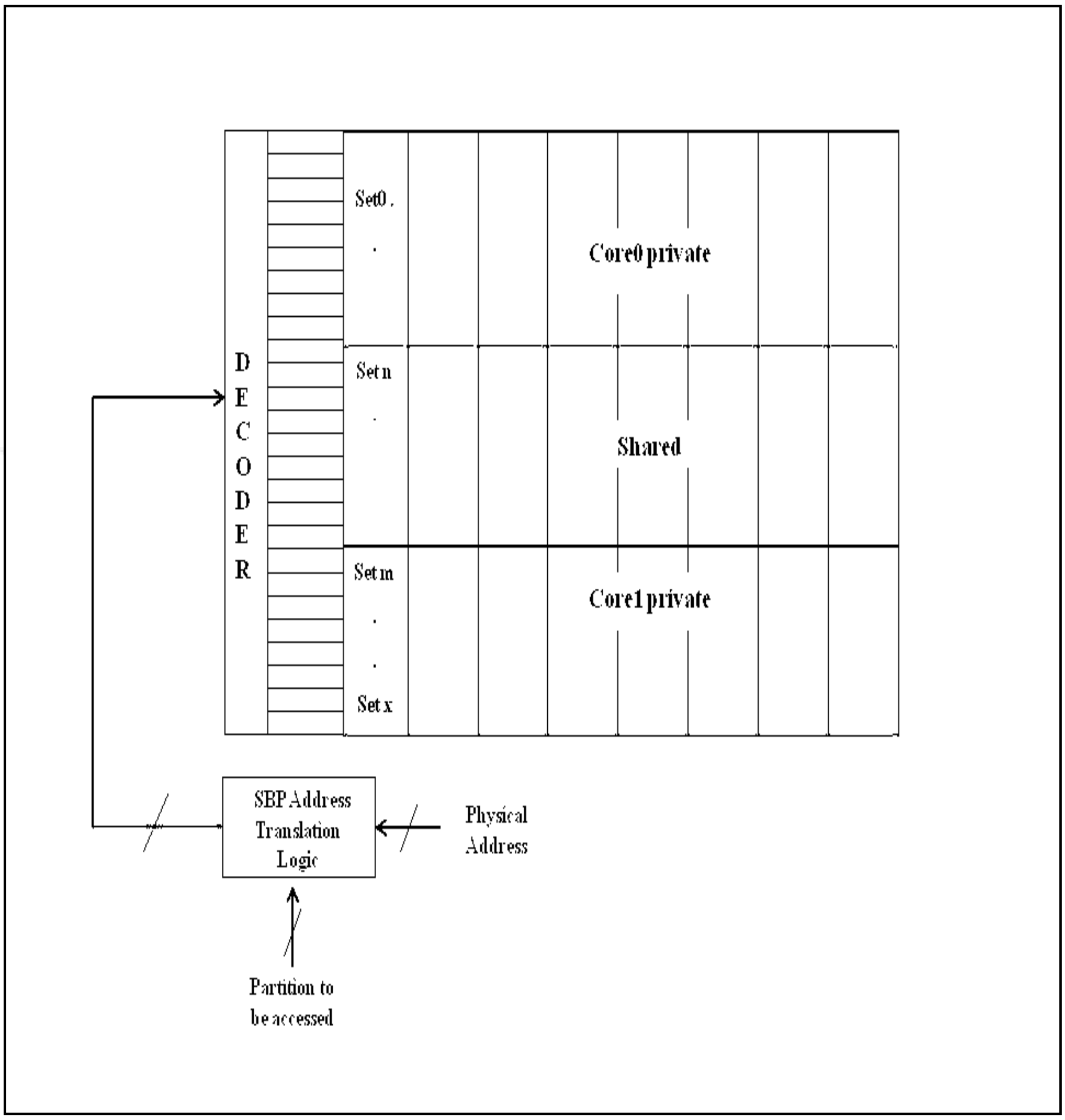

Figure 3.3. Level two partitioned cache for two cores

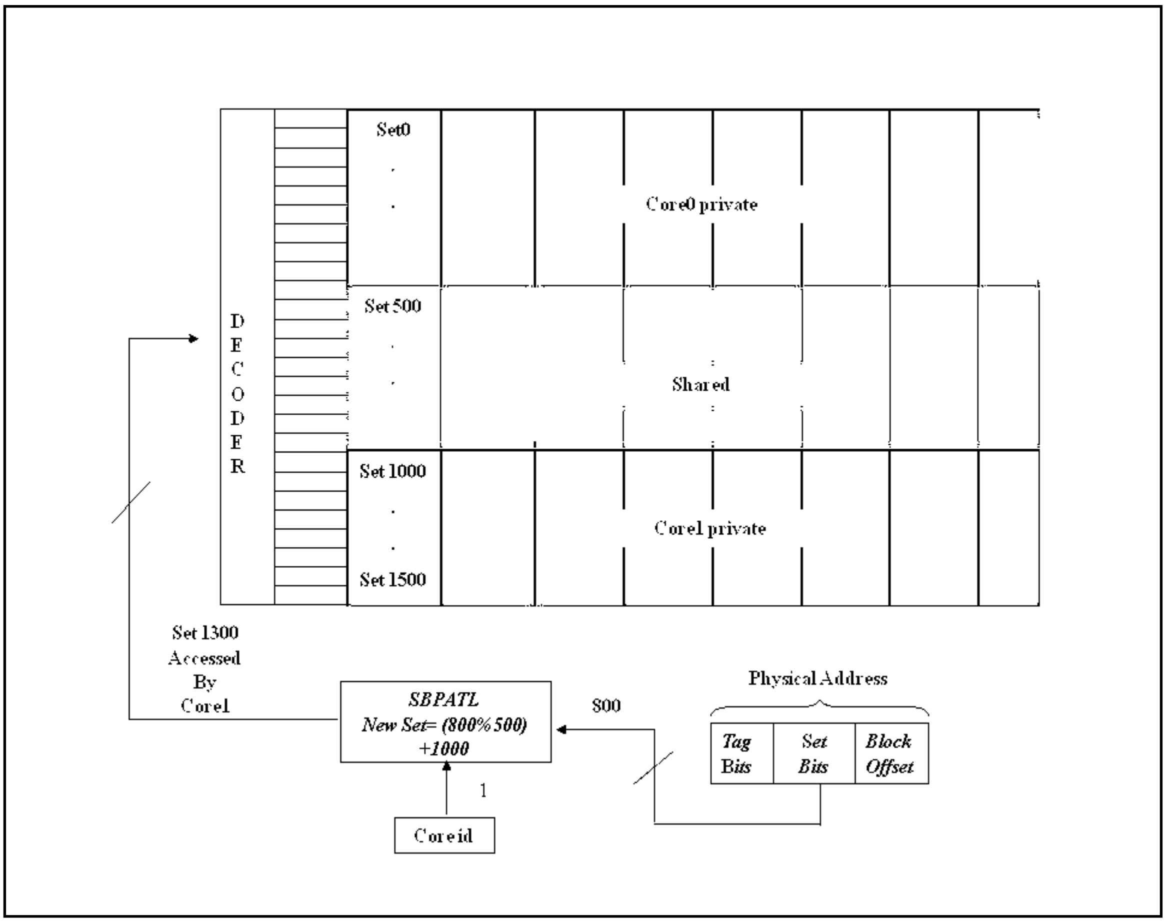

Figure 3.4. SBPATL example in SBP cache

3.2. SBP THRESHOLD VALUE

Our partitioning algorithm mainly depends on the miss rate difference between cores. In our case we need to define a suitable miss rate difference threshold value, which is tailored to the miss rate values of applications, in order to keep track of application behaviors.

The chosen miss rate difference threshold value is either fixed or dynamic. In our study, first we tested five per cent and 10 per cent as fixed threshold values to decide whether a core needs more cache sets or not in our partitioning algorithm. Unfortunately, the performance results were inconsistent for these fixed threshold values as we expected. As a result, we decide utilizing a dynamic threshold mechanism called SBPMDTA.

3.2.1. SBP miss rate difference threshold average (SPBMDTA)

SBPMDTA parameter monitors or keeps the history of the miss rate difference values among cores for a specific window. We suppose if we can watch the miss rate difference values among cores for multiple periods, we can reach to a good estimation of the threshold value that later allows us to decide whether an application demands additional cache space or not. The calculation of SBPMDTA is done by adding a queue structure that keeps the value of dynamic miss rate difference values among cores of each period. In the next step, we can get SBPMDTA by summing all the values of the queue elements (*Di)* divided by the number of elements in the queue (*n*) see equation (3.4). If the queue size is chosen to be a power of two then the division may be calculated by simple shift operations.

$$SBPMDTA = \frac{\sum_0^n Di}{n} \tag{3.4}$$

3.3. HL2UC ACCESS PROTOCOL

In our design, cores access the HL2UC searching for a specific data block. Here, the search process is different than that of a second level shared cache. On each cache access, a core first searches its own private partition in the HL2UC, if the required data block is there. On a hit, the SL2UC hit time is returned. On a miss, the shared partition in the HL2UC is searched. When the data block is located on that partition, a hit time, which is equal to the sum of the hit time of the private partition and the hit time of the shared partition, is returned. The hit times for both partitions are equal. This is due to accessing the same physical resource twice. When the data block cannot be located in the shared partition, an access to the off-chip memory is carried out, and the data block is read to both shared and private partitions of the HL2UC. Figure 3.5 shows the HL2UC access protocol flowchart.

3.4. HARDWARE IMPLEMENTATION AND COMPLEXITY OVERHEAD

The hardware implementation of SBP mechanism is shown in Figure 3.5. In order to estimate the complexity overhead, we calculate the approximated number of transistors for each hardware element that is used in our implementation. After, calculating the total number of transistors that used to build two MB shared cache, we find that the hardware implementation of our method requires negligible complexity increase of about 0.01 per cent.

Transistor numbers of each element in the SBP method hardware implementation with the number of transistors of used cache space are shown in table 3.1 and 3.2.

Table 3.1. SBP mechanism elements cost

SBP mechanism elements	**Bit numbers in each element**	**Transistor numbers in each element**
Four miss and access counters	64	(4*64*6) =1536
Two division units	64	6912 = (2*64*54)
One division unit	7	378 = (7*54)
One multiplication unit	20	1080 = (20*54)
Two subtraction units	7	392 = (2*7*28)
One adder unit	7	196 = (7*28)
Three latches	7	126 = (3*7*6)
Select inputs	5	30 = (5*6)
Shift registers assuming D f/f	7	1260 = (7*5*36)
One comparator unit	7	147 = (7*21)
Total SBP mechanism cost:	12057 transistors	

Table 3.2. HL2UC cost in the SBP mechanism

HL2UC elements	**Bit numbers in each element**	**Transistor numbers in each element**
2048 set	20	120 = (20*6)
8 ways	3	18 = (3*6)
Data block length	128	768 = (128*6)
Tag bits	10	983040= (10*2048*8*6)
Dirty bit	1	98304 = (2048*8*6)
Valid bit	1	98304 = (2048*8*6)
Total HL2UC cost:	1180554 transistors	

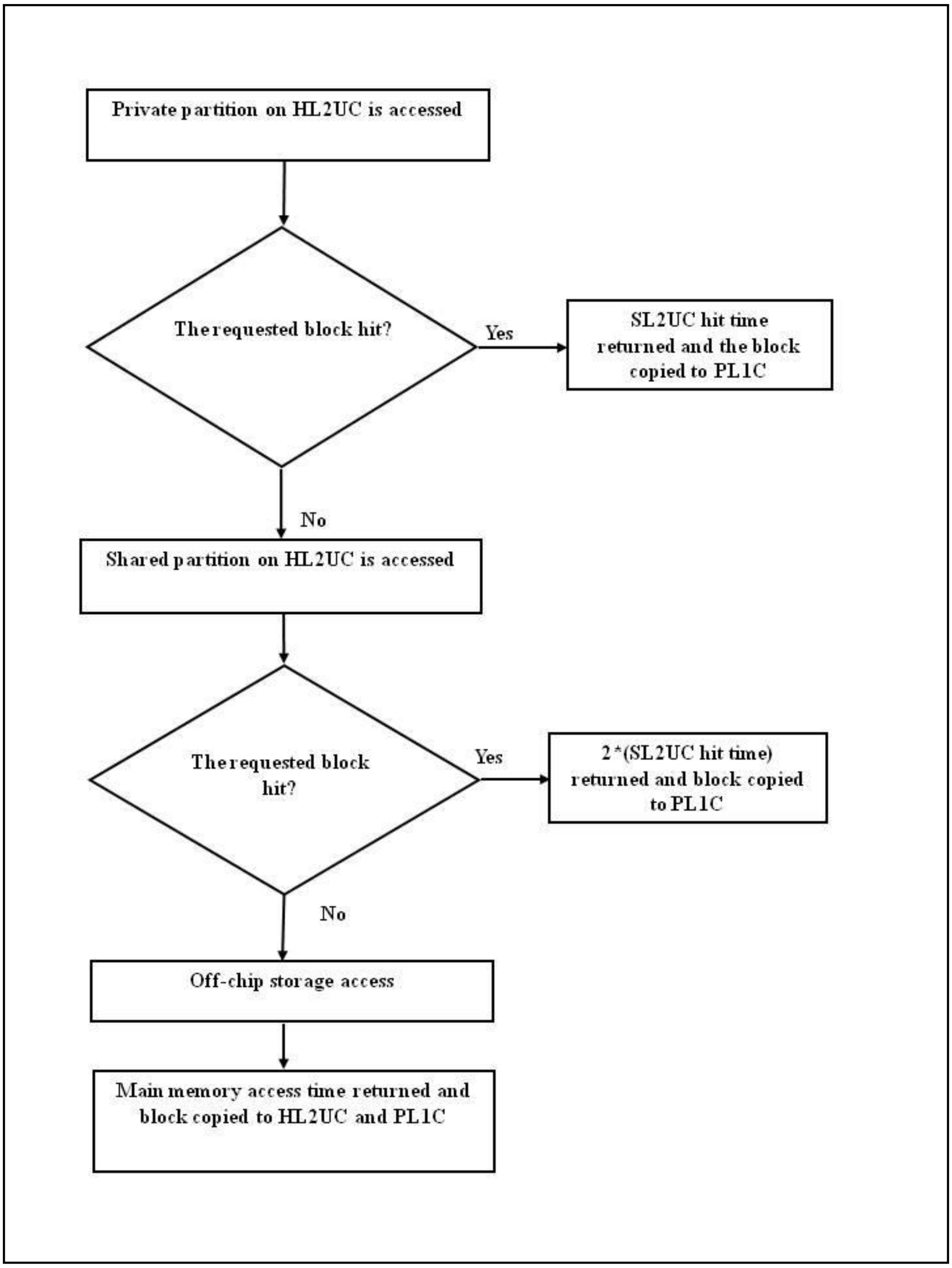
Private partition on HL2UC is accessed
The requested block hit?
Yes
SL2UC hit time returned and the block copied to PL1C
No
Shared partition on HL2UC is accessed
The requested block hit?
Yes
2*(SL2UC hit time) returned and block copied to PL1C
No
Off-chip storage access
Main memory access time returned and block copied to HL2UC and PL1C

Figure 3.5. HL2UC search protocol flowchart

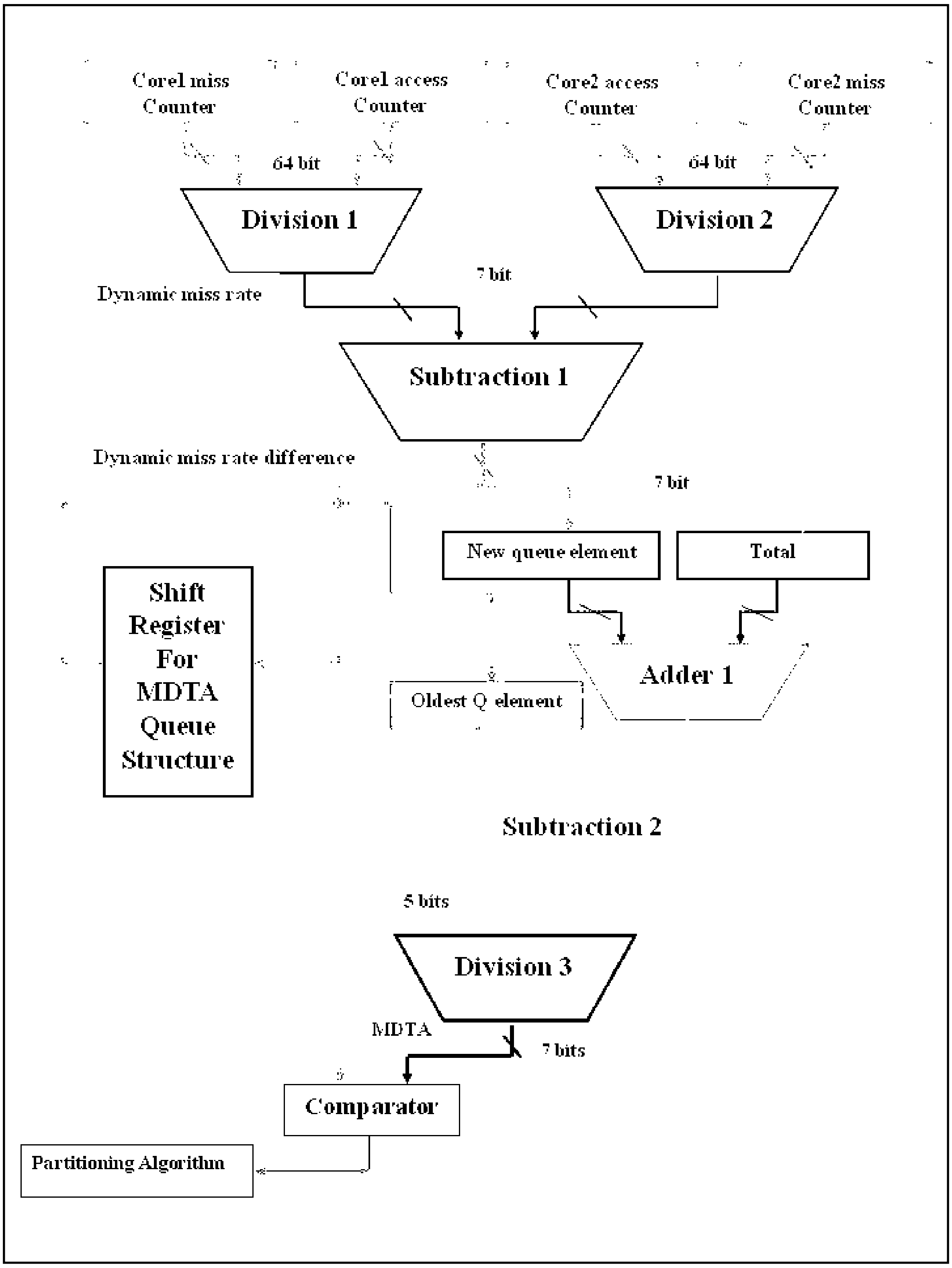

Figure 3.6. Hardware implementation of SBP cache

4. EXPERIMENTAL METHODOLOGY

In order to evaluate our proposed design we used M-sim [14], a detailed multithreading simulation environment that also includes a CMP model. In our study, we simulated two cores with identical specifications. We only change the cache parameters throughout our experimental study and the rest of the parameters of the processors are kept at default. These parameters are shown in table 4.1.

Table 4.1. Processor core specifications

Maximum number of instructions to execute	200 Million
Number of instructions skipped before timing starts	50 Million
Number of contexts allowed per core	1
Instruction decode B/W (instructions/cycle)	8
Out of order instruction issue width B/W (instruction/cycle)	8
Issue queue (IQ) size	64
Instruction commit width B/W (instrs/cycle)	8
Load/Store queue (LSQ) size	48
Reorder buffer (ROB) size	256
Physical register file (RF) size	256
Total number of integer ALUs	8
Total number of integer multiplier/dividers	3
Total number of floating point ALUs	8
Total number of floating point multiplier/dividers	3

4.1. BASE LINE MEMORY CONFIGURATION

Each processor core utilizes a first level private instruction cache and a data cache. Second level cache is shared between the cores. Detailed memory specifications are given in table 4.2.

Table 4.2. Memory specifications

Private L1 Instruction-cache and Data-cache configuration	32KB, 32B block size, 512 sets, 2-way with LRU replacement policy
Unified shared level2 cache configuration	2MB, 128B block size, 2048 sets, 8-way with LRU replacement policy
L1 I-cache and D-cache hit time	1 cycle
L2 shared cache hit time	20 cycles
Memory access latency	300 cycles
Memory access bus width (in bytes)	8
Total number of memory system ports available	2
Number of sets added/removed (resizing amount) for resizing (*s*)	2,4,8,128
update interval	(100k, 1M, 3M, 5M, 15M)cycles
Number of SBPMDTA queue entries (*n*)	5, 10, 25

The last three rows values in table 4.2 can be considered as the default values of SBP cache configuration, other values for these parameters are tested and the results will be briefly discussed in chapter five.

4.2. BENCHMARKS

We use Spec2K benchmarks to evaluate our work [15]. Since the second level cache is shared between two cores, we need to use a mixture of two applications. To cover wide range of mixtures in our work, we classify the workloads into four categories in terms of their characteristics:

- Computation-intensive workloads (C).
- Memory-intensive workloads (M).
- Hybrid workloads (H1) containing one computation-intensive and one memory-intensive application.
- Hybrid workloads (H2) containing one memory-intensive and one balanced application. In balanced applications, both computation- and memory-intensive behavior is observed in a single run.

As a result, a collection of 20 application mixtures is composed. Table 4.3 gives the details of these mixtures.

There are two types of Spec2K benchmarks table 4.4 and table 4.5 shows the details of the simulated spec2k benchmarks. Using the simulator, we ran each benchmark alone for 200 million instructions, and we recorded the load/store instruction percentage, Figure 4.1 shows these results in detail. Finally, we categorized these benchmarks based on the obtained percentages. We assumed that computational benchmarks have a load/store percentage of less than 40 per cent, memory-intensive benchmarks have a load/store percentage of more than 45 per cent, and hybrid workloads have a load/store percentage of between 40 per cent and 45 per cent.

Table 4.3. Workloads type

CATEGORY	***WORKLOAD BENCHMARK1,BENCHMARK2***
TYPE C (C,C)	AMMP,SWIM
	FMA3D,PERLBMK
	WUPWISE,SIXTRACK
	WUPWISE,PERLBMK
	SWIM,FMA3D
TYPE M (M,M)	VPR,APSI
	TWOLF,BZIP2
	PARSER,TWOLF
	MGRID,APSI
	MESA,MGRID
TYPE H1(C,M)	FMA3D,MGRID
	PERLBMK,PARSER
	PARSER,FMA3D
	GZIP,PARSER
	SWIM,TWOLF
	TWOLF,AMMP
TYPE H2(M,H)	MGRID,ART
	ART,MESA
	APSI,ART
	PARSER,ART

Table 4.4. Integer benchmarks

Benchmark name	**Benchmark general description**
Gzip	Compression
Vpr	FPGA Circuit Placement and Routing
Parser	Word Processing
Perlbmk	PERL Programming Language
Twolf	Place and route simulator

Table 4.5. Floating point benchmarks

Benchmark name	**Benchmark general description**
Wupwise	Physics / Quantum Chromo dynamics
Swim	Shallow Water Modeling
Mgrid	Multi-grid Solver: 3D Potential Field
Mesa	3-D Graphics Library
Art	Image Recognition / Neural Networks
Ammp	Computational Chemistry
Fma3d	Finite-element Crash Simulation
Sixtrack	High Energy Nuclear /Physics Accelerator Design
Apsi	Meteorology/ Pollutant Distribution

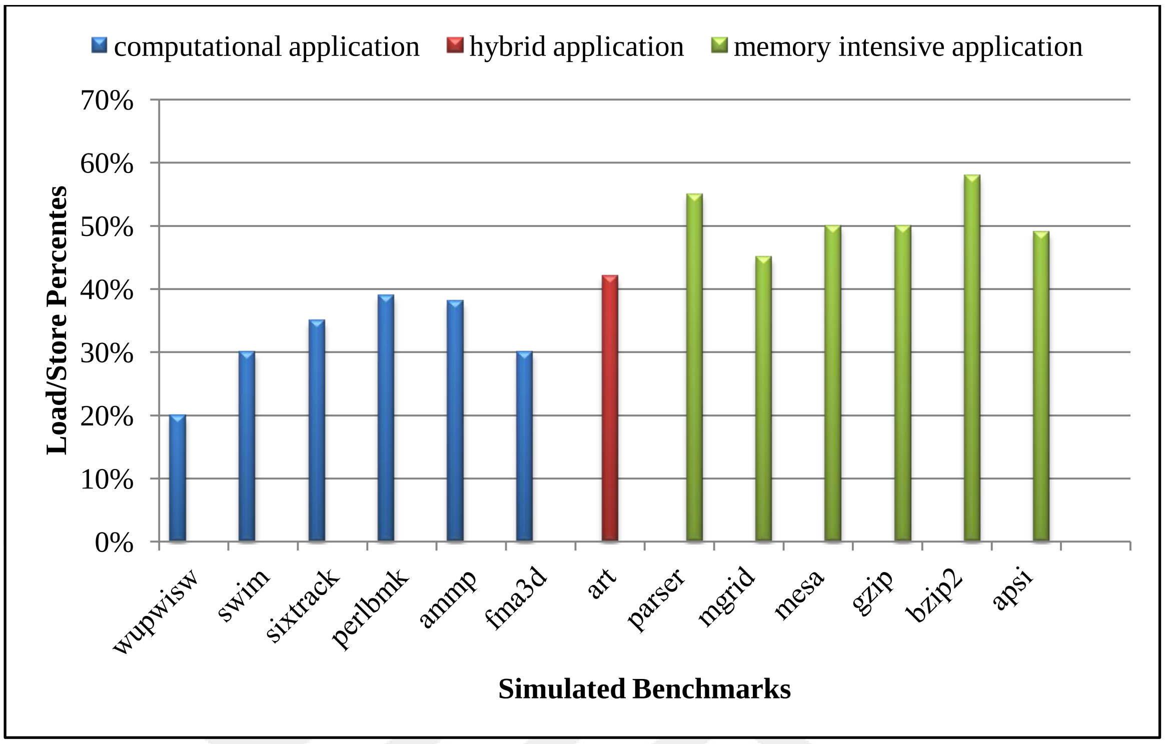

Figure 4.1. Load/store percentages for simulated benchmarks

5. SBP CACHE SIMULATION RESULTS

The performance results obtained from our design are compared to two different cache partitioning configurations. The first configuration is the baseline shared cache among competing applications and the second configuration is the static or fixed cache partitioning configuration which allocates fixed amount of cache sets to each application.

The disadvantage of shared cache compared to fixed partitioned cache the data pollution, since it is accessed by all cores. Additionally, shared cache configuration can be unfair to running threads, especially when one of the running applications is a computational-intensive and the other is a memory-intensive application. In that case the memory-intensive application can steal cache sets from the computational-intensive application. The fixed partitioned cache configuration; on the other hand provides dedicated area for each core to prevent data pollution. However, giving a fixed amount of space to each running application is not a good idea, since an application cannot request more cache resource.

We need to implement the fixed cache partitioning method based on sets in order to compare its performance results with our SBP cache results, since the fixed cache partition based on sets not supported in the used simulator. The implementation steps for the fixed partitioning are as follows:

- Divide the cache into three fixed-size region with each core having its own private area and a shared area for both cores.
- Mapping all data references of a core is supported by the fixed address translation logic (FATL), which has a similar function of the (SBPATL). The main difference between FATL and SBPATL the partition size fixed (partitioned to be accessed by specified core id). Remember that in SBPATL the partition sizes are dynamically changed based on the miss rate of the running applications.
- Accessing the new set number provided by the FATL results in either hit or a miss. In case of a data block hit the returned latency will be equal to the hit time of the

level two shared cache which is 20 cycles in our simulation; otherwise, a new set from the shared region will be provided by the FATL.

- The given set will be accessed in the level two shared area and on a data block hit, the returned latency will be equal to 40 cycles (20 cycles in PL2+ 20 cycles in SL2) since, we are accessing the same physical resource twice. Finally on a data block miss in the SL2 area, an off-chip access is required and the returned latency will be 300 cycles.

5.1. SBP CACHE WITH DIFFERENT UPDATE INTERVALS

The *update interval* parameter specifies the period that we run our partitioning algorithm to dynamically determine which application needs more private space in the SBP cache. We test different values of the *update interval* parameter and compare its effect on the performance results. Figure 5.1 shows these results, in detail.

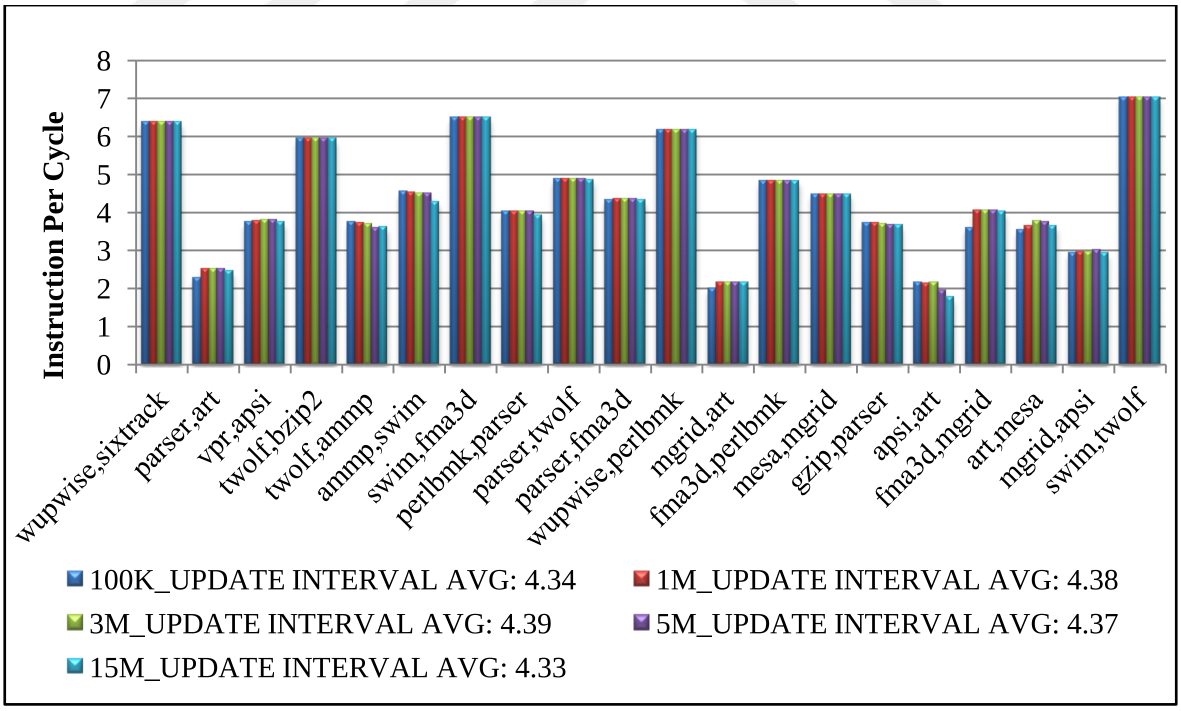

Figure 5.1. SBP cache performance with different *update intervals*

The performance results from Figure 5.1 shows that SBP cache with *update interval* of one and three million cycles performs 1.1 per cent on the average better than with small *update interval value* (100 Kcycles) such as (parser,art), (fma3d,mgrid) and (mgrid,art). These results show that when the partitioning algorithm is run too often, the performance degradation is expected. We can explain this phenomenon as follows: when the application behaviour is stable we should avoid cache flushes as much as possible. However, when the partitions are resized the cache partitions are mandatorily flushed resulting loss of precious data and cache hits in those partitions.

Meanwhile, when large *update interval*s are considered, such as five and 15 Mcycles, 1.6 per cent average performance reduction is also observed. In some mixtures, this performance degradation is quite severe. For example, in (ammp,swim) and (apsi,art), the performance drop is 5.7 per cent and 17 per cent, respectively. The main reason of this performance degradation can be explained as follows: with large *update interval* values the partitioning algorithm may not adapt to the rapid changes in applications' behaviours.

When an application's behaviour is changed, but if the partition sizes are not adapted to the most recent needs of that application, the cache miss rate will increase resulting in performance reduction. In the extreme case when the *update interval* is chosen to be a higher value the performance of the SBP cache will be close to the performance of a fixed partition cache configuration.

5.2. SBP CACHE AND THE RESIZING AMOUNT

At this point we want to emphasize that, our main goal in this thesis is to logically and dynamically partition the cache based on sets. So instead of giving cache ways to the required application and sacrificing the fine-grain of control, we give specific number of sets to the application with higher miss rate. Here we study the impact of resizing amount, the number of sets that are added or removed to each cache partition. Figure 5.2 shows the performance results of SBP cache with two, four, eight and 128 sets of resizing amounts with one million cycles *update interval*.

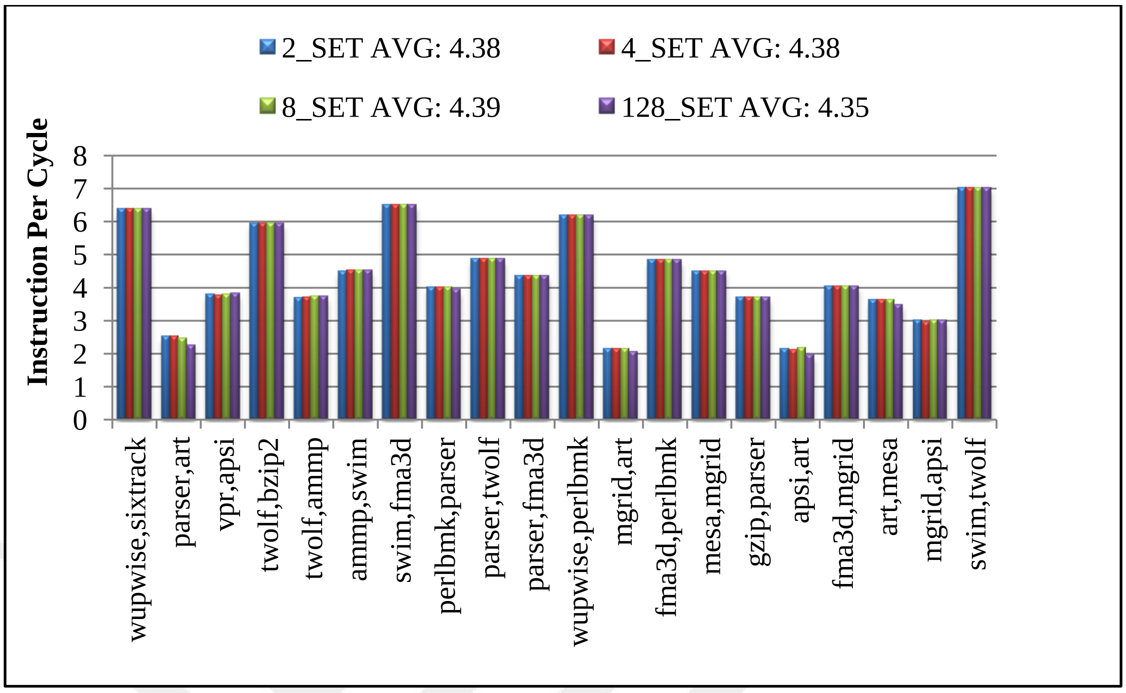

Figure 5.2. SBP cache performance with various resizing amounts

The results shown in Figure 5.2 imply the power of the set based cache partitioning. Here when the resizing amount of 128 sets is considered, that amount is equal to the minimum resizing amount of a way partitioning configuration with 16 cache ways. As a result, we can compare our SBP cache with a way partitioning cache.

In hybrid workloads, the SBP cache partitioning algorithm performs better than the way partitioning algorithm by two per cent on the average and by 0.7 per cent in the all simulated benchmarks. Here, clear performance drops in the way partitioning algorithm by 10 per cent in (parser,art) and 4.2 per cent in (mgrid,art) mixtures are observed. The performance reduction is related to granting an application more space than its need which later causes an oscillation in the partitioning control mechanism. The oscillation depends on the given resizing amount to the demanding application. In this case, giving large amount of space to an application with little space requirements, such as art, will result in increasing the corresponding miss rate for a memory-intensive application like parser. Later, this increase in the cache miss rate will badly affect the decision process of the partitioning algorithm for the next *update interval*.

As a result, the given cache resource will be taken from art and be dedicated to parser. These oscillations in the control mechanisms will continue for many *update intervals*, resulting performance degradation on the processor.

In contrast, way partitioning algorithm performs better than SBP cache in some memory-intensive workloads. For example, performance drops in the SBP cache by 1.1 per cent in (vpr,apsi) and 0.4 per cent in (mesa,mgrid) mixtures are observed. When memory-intensive application momentarily needs large number of additional cache data blocks, then the SBP cache with 128 set resizing amount (or the way partitioning mechanism, since they supply nearly the same amount of cache resource) may perform better than other configurations with smaller set resizing amounts, since they can supply the required amount of cache resources, instantly.

5.3. SBPMDTA QUEUE ENTRIES EFFECT ON SBP CACHE

In chapter three, we explained SBPMDTA parameter and its role in the SBP cache. Then we showed that it can be implemented by a queue structure that keeps the miss rate difference between the running applications (see equation (3.4)). Here, we study the influence of the queue size (the number of the queue entries (*n*)) on the performance results of the SBP cache.

In Figure 5.3 the performance results of the SBP cache with SBPMDTA queue entries equal to five, 10 and 25 elements are shown. Relative to SBP cache performance with five SBPMDTA queue entries, a performance reduction by 3.3 per cent in the hybrid workloads and by 1.5 per cent in the all simulated benchmarks are observed, when 25 SBPMDTA queue entries are considered.

We believe this reduction in the performance results for hybrid workloads when the number of queue entries are equal to 25 elements is due to applications which contain multiple active working sets, since each application references certain data blocks in the cache, the active working set may be changed throughout the life time of an application which later results in different values for the cache miss rate in different periods.

In addition, this change in the miss rate will reflect its impact on the determined value of SBPMDTA. Using large queue sizes will results in keeping long history of the miss rate difference.

Later, the obtained SBPMDTA value will not reflect the periodic changes in the application behaviour. This cumulative SBPMDTA value will prevent the application with moderate cache space requirement to satisfy its momentary resource requirements, and, as a result, performance degradation will occur. In Figure 5.4, we see an example of such degradation in (parser,art) mixture. When the simulation time is equal to 20 and 67 Mcycles, the application *art* cannot get all the resource it requires from the cache space when the number of queue entries is equal to 25 elements. Since, the obtained SBPMDTA value is greater than dynamic miss rate of *art*, and as a consequence, the SBP cache control mechanism refuses to satisfy *art*'s request. Meanwhile, *art*'s request is satisfied when the queue size of five entries is utilize, since the SBPMDTA value reflects a portion of *art*'s working set history.

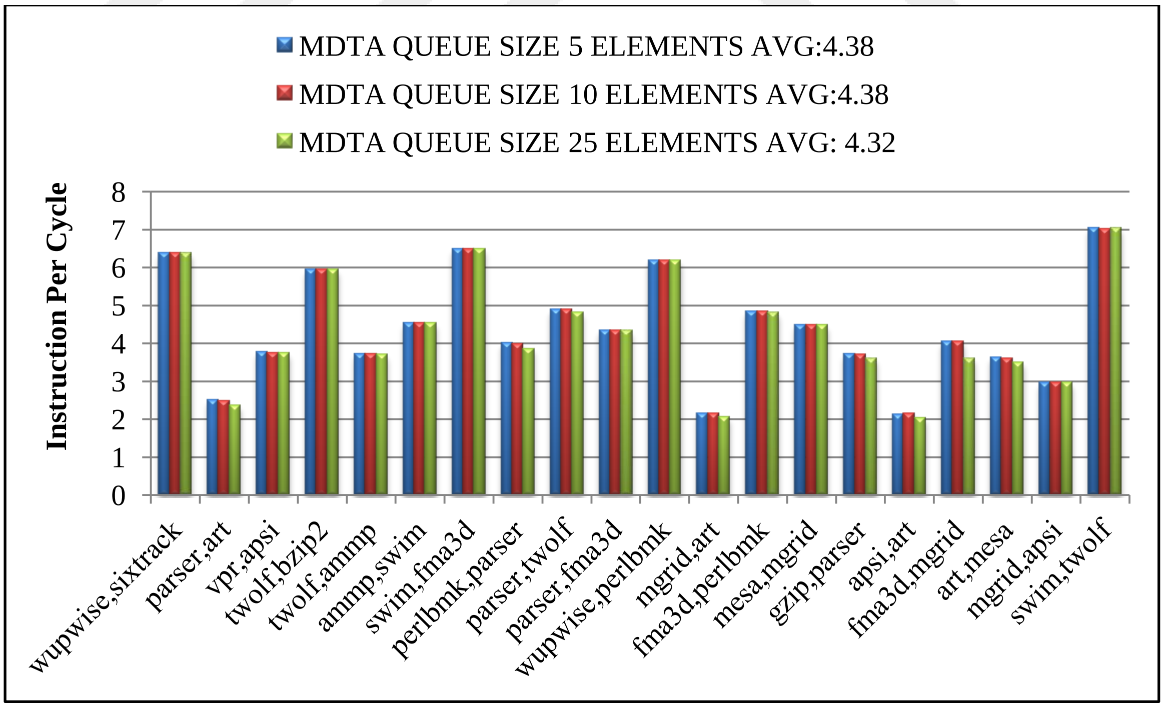

Figure 5.3. SBP cache performance with different queue size

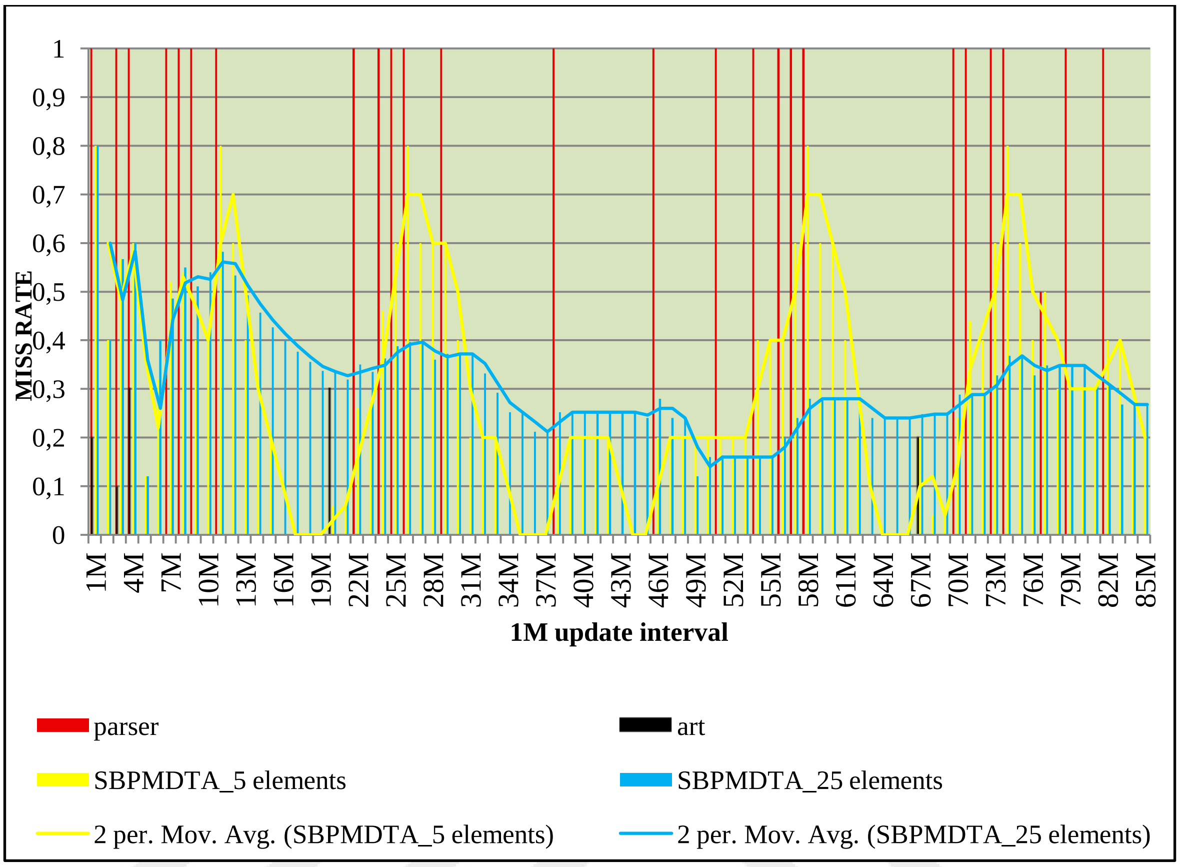

Figure 5.4. SBPMDTA with 5 and 25 queue entries

5.4. SBP CACHE RESULTS FOR DIFFERENT WORKLOADS

5.4.1. SBP cache and C-type workloads

The performance results of the SBP cache, shared cache and fixed partition are very close to each other in all c-type workloads. This is an expected result, since these workloads do not heavily depend on memory accesses, and the given cache space is enough for these applications to achieve good performance. Figure 5.5 shows the performance results for the simulated configurations.

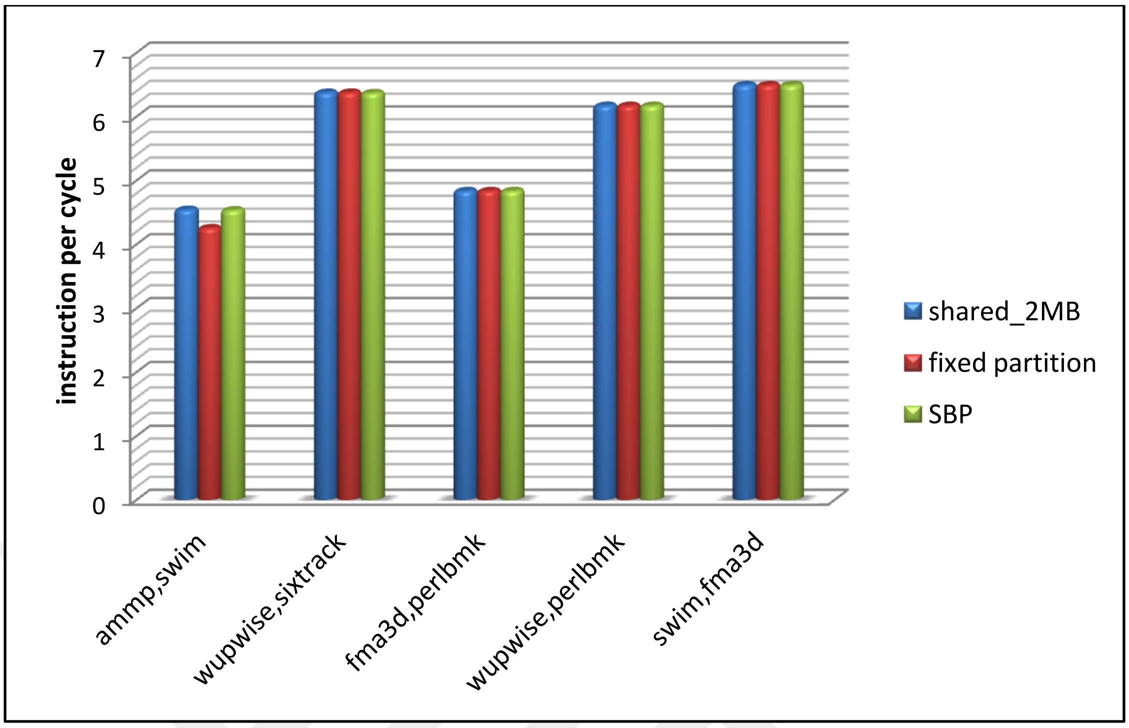

Figure 5.5. Performance of SL2, fixed partition and SBP
For C-type workloads

5.4.2. SBP cache and M-type workloads

In contrast to C-type workloads; M-type workloads require an efficient cache configuration tailored to their needs. Here, our partitioning algorithm and fixed cache partitioning have closer performance results, since they both reduce the effect of data pollution by dedicating a private area for each running application. The SBP cache performance gain in M-type workloads is nine per cent.

In some workloads such as (vpr,apsi) and (mgrid,apsi) mixtures, we notice a performance improvement by 8.3 per cent and 24 per cent respectively compared to the fixed partitioning case, since our algorithm provides additional cache sets to the applications with higher memory demands. Figure 5.6 shows the performance results for all three cache configurations.

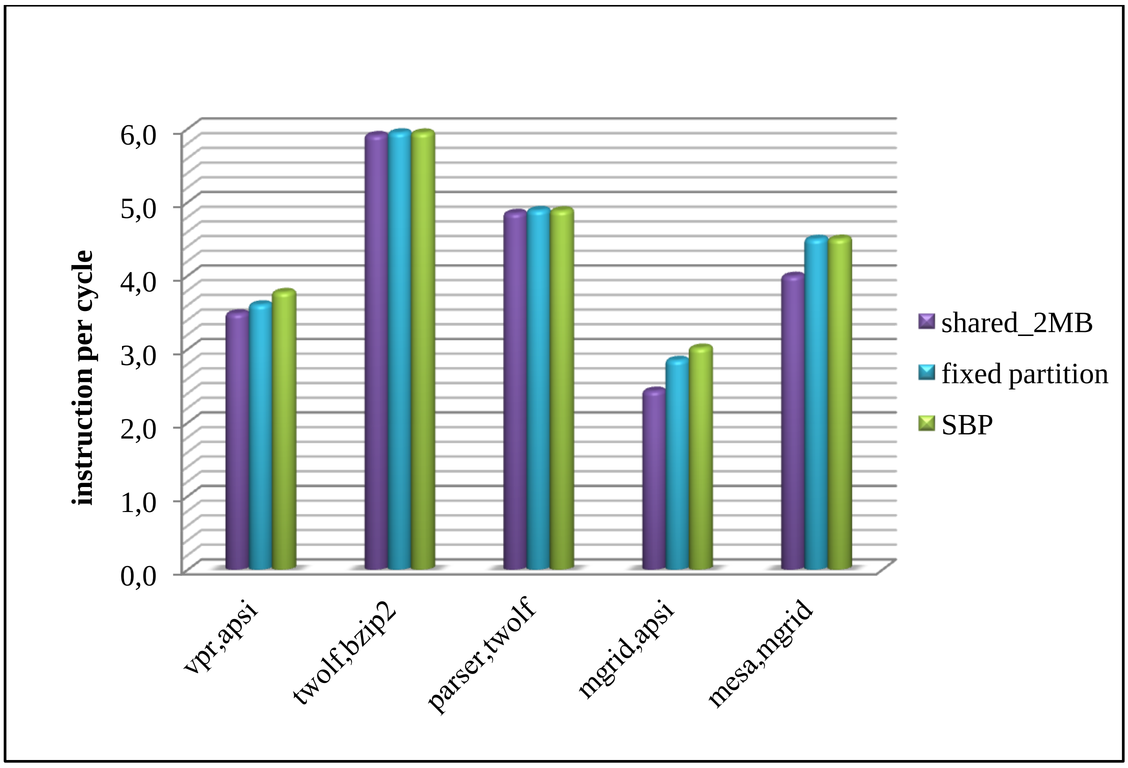

Figure 5.6. Performance of SL2, fixed partition and SBP
For M-type workloads

5.4.3. SBP cache with H1-type workloads

Figure 5.7 shows the results of H1-type workloads. In these workloads the SBP cache performs better than the shared cache configuration by 2.4 per cent. However, the performance results of the fixed cache partitioning are the best for both (perlbmk,parser) and (fma3d,mgrid) mixtures.

Lower performance of SBP cache for these workloads can be explained with the cache invalidation problem. When assigning additional sets to the required cores, the whole data blocks in the private and shared partitions are invalidated. The impacts of these invalidations are further studied in section 5.5.

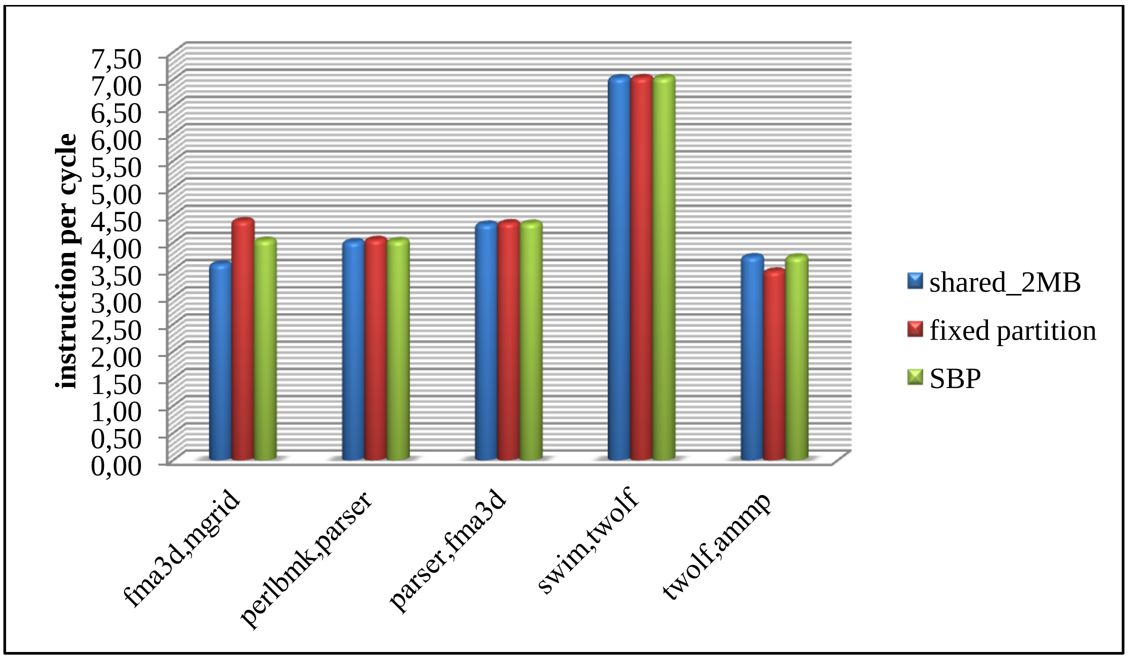

Figure 5.7. Performance of SL2, fixed partitioned and SBP
For H1-Type workloads

5.4.4. SBP cache with H2-type workloads

In Figure 5.8, H2-type workload results are shown. Here, the SBP cache outperforms better than other configurations in all simulated workloads. The (apsi,art) mixture in that graph shows that the fixed partitioning performs worse than the shared configuration. This is quite possible especially when one of the workloads starts experiencing capacity misses. Again, our SBP cache supplies the right amount of partition size to each core, and achieves the best performance results. 29.4 per cent performance gain is achieved by SBP cache for H2-type workloads.

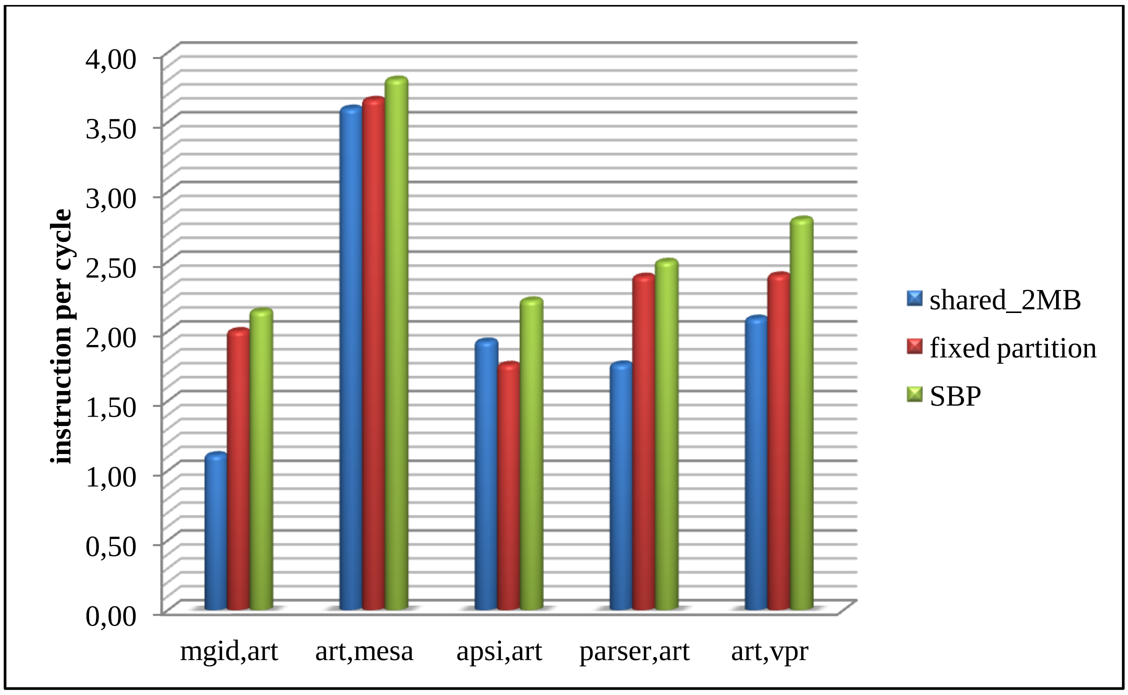

Figure 5.8. Performance of SL2, fixed partition and SBP
For H2-Type workloads

5.5. IMPACT OF CACHE INVALIDATIONS ON SBP CACHE

We studied the performance impact of cache invalidations due to resizing of the cache partitions. In our study, we assume write-back write policy is used in the baseline and the SBP cache. As a result, when a partition size is to be changed, all the dirty data blocks in that cache partition (or partitions since other partition sizes may also be affected from that change) must be moved to lower memory structures in the memory hierarchy.

Figure 5.9 shows the performance difference between the standard SBP cache with write-back policy and the SBP cache with write-through policy (SBPWT) which does not incur any invalidation penalty. After running all mixtures, the obtained results show that if the cache invalidation problem is solved, the performance of the SBP cache will improve by only 0.3 per cent, which can be considered negligible. Also note that this performance improvement comes with an additional power cost due to write-through cache policy, and therefore, we show that SBPWT configuration is not very feasible.

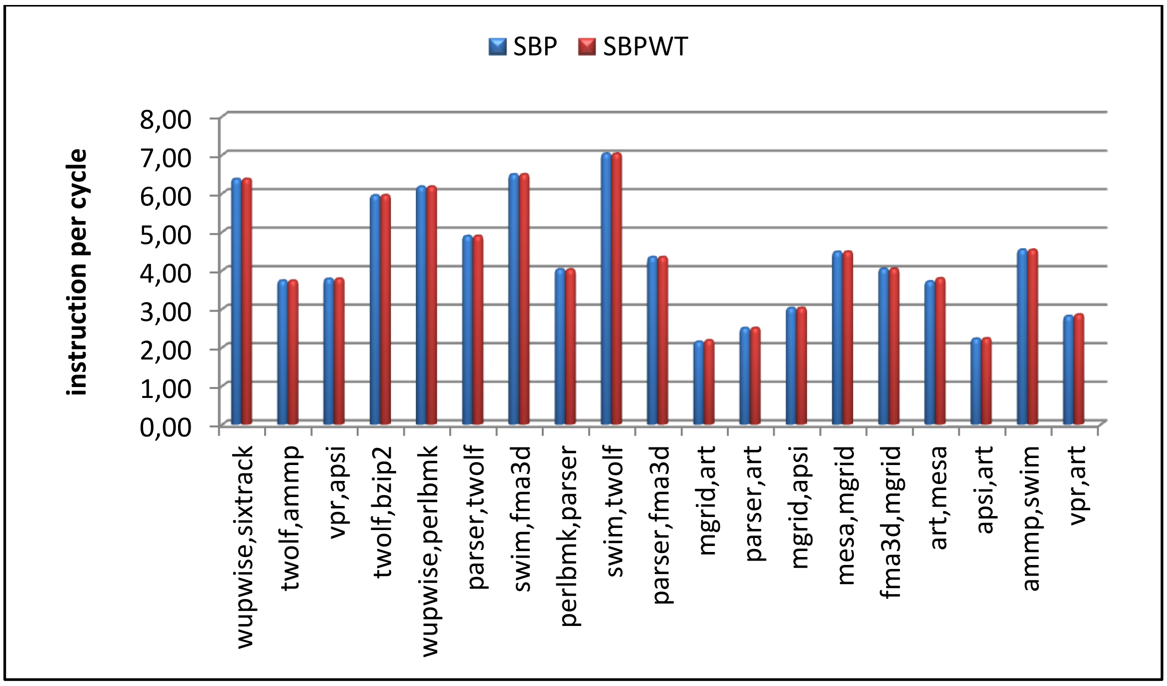

Figure 5.9. Impact of cache invalidations on SBP performance

5.6. EFFECT OF SHARED PARTITION SIZE ON SBP CACHE

The performance results for various initial sizes of the shared partition were examined. Figure 5.10 shows the collected performance results of SBP cache for 1024, 24 and four sets of shared partition sizes.

When the initial shared partition size is set to four, compared to 1024 sets, the performance of the SBP cache for type H2 workloads decreases as in (mgrid,art) and (parser,art) mixtures.

These low performance results are expected because the memory-intensive application (mgrid or parser) requires more cache partition than the hybrid (equally memory- and computation-intensive) application in the mixture. After a short execution period, the shared partition is consumed, and the memory-intensive application starts stealing cache sets from the private partition of the hybrid application resulting in degradation in the overall performance.

When we consider the H1-type workloads similar to (fma3d,mgrid), the memory-intensive application may still steal cache sets from the private partition of the computation-intensive application but, as we expected, this does not affect the performance degradation in that application. The results shows three per cent performance improvement in the memory-intensive application and stable performance in the computation-intensive application compared to the configuration with initial shared partition size of 1024 sets.

Furthermore, when the initial size of the shared partition is set to 24 sets, we measured a performance increase by four per cent in hybrid type workloads such as (mesa,art). This is because the memory-intensive application steals cache sets from the shared partition but not from the other private partition.

Additionally, we measured performance improvement compared to the configuration with initial shared partition size of 1024 sets in memory-intensive mixtures such as (mesa,mgrid). Here, both applications start with large private partitions, and then compete to get more sets from the shared partition without stealing cache sets from each other. As a result, performance of each application can be improved in such workloads.

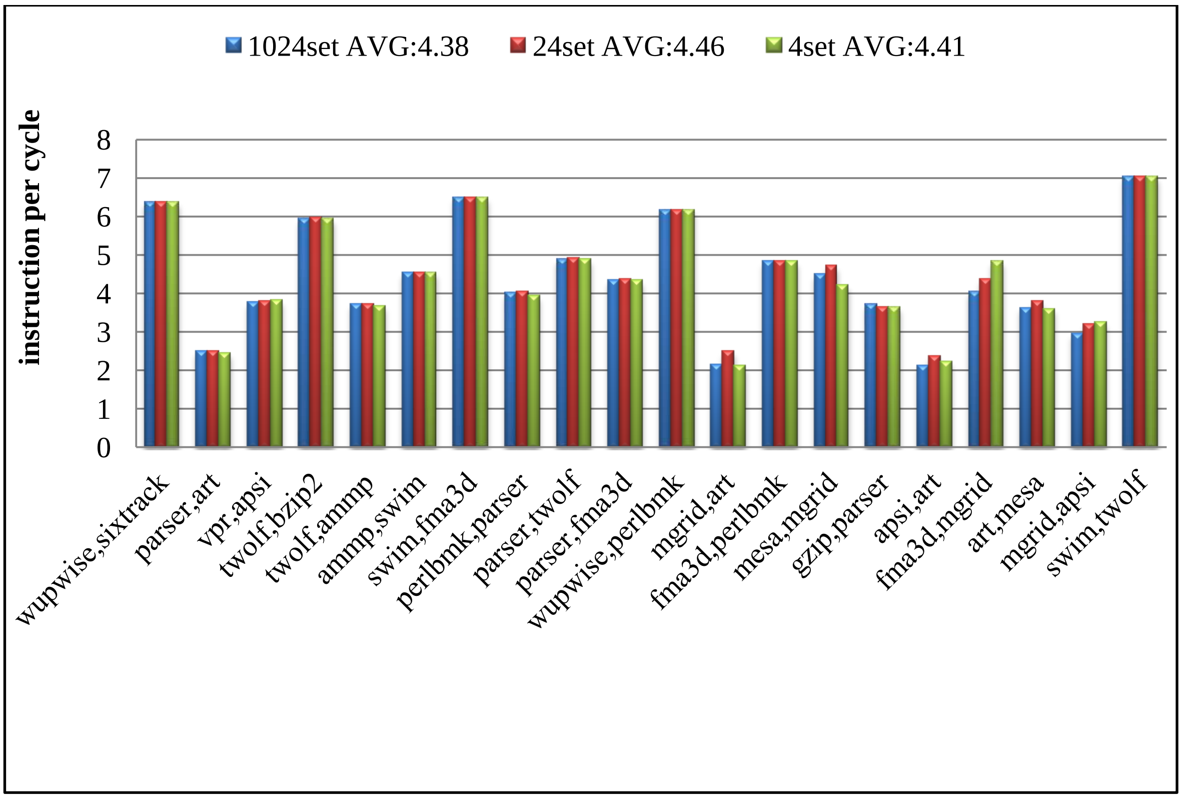

Figure 5.10. SBP performance with different shared partition size

6. CONCLUSIONS AND FUTURE WORK

The traditional configurations of the second level cache as either shared or private do not help optimizing the cache performance of CMPs. We believe that using a dynamically partitioned hybrid cache configuration gives better performance results in CMPs by reducing the number of misses for each core.

The SBP cache, proposed in this study, is a fine-grain adaptive method for partitioning L2 cache based on sets. This method is suitable for workloads with or without sharing. Moreover, the hardware implementation requires negligible complexity increase, which is as much as 0.01 per cent.

The SBP cache improves the overall performance of memory-intensive application mixtures by nine per cent and of hybrid workloads by more than 15 per cent, on the average across all simulated benchmarks. Since the SBP cache changes the physical addresses to set address mappings, after each cache resize operation the old cache entries become no longer accessible. However, the performance penalty resulting from those invalidations is negligible.

The *update interval* parameter in the SBP cache must not be chosen very large or very small. Since choosing it very large makes SBP cache similar to a fixed cache configuration, in contrast, choosing the *update interval* very small results in continuous precious data flushing. Moreover, by assigning 128 set resizing amount value, we actually get a chance to compare the SBP cache with the way partitioning method. An average performance reduction of a way partitioning method by 0.4 per cent and two per cent are observed in the all simulated benchmarks and in the hybrid workloads, respectively. On the other hand, for some of the memory-intensive workloads, performance degradation in the SBP cache is observed ((vpr,apsi) 1.1 per cent and (mesa,mgrid) 0.4 per cent), since the SBP cache cannot supply the required amount of cache resources, instantly.

Furthermore, we study the effect of various SBPMDTA queue sizes on the SBP cache performance. We found that by assigning large queue sizes will not improve the SBP cache performance. In contrast, a performance reduction by 3.3 per cent in the hybrid workloads and 1.5 per cent in the all simulated benchmarks are observed.

The shared partition size also affects the SBP cache performance. Here, choosing small shared partition may lead to performance degradation especially in the memory-intensive workloads. Since these workloads need large cache space to improve their performance, and reducing the shared partition size increases the cache steals among running applications which results in a larger number of cache flushes.

The SBP cache in CMPs with two cores gives promising results in term of performance gains for private workloads. As future work there are additional requirements needed to test the effective of SBP cache in CMPs which are as follows:

- Test SBP cache on shared workloads.
- Introduce other metrics like IPC (for fairness) combined with the miss rate to dynamically allocate partitions to cores.
- Mechanism for deciding workload type (shared, private).
- Design and test the SBP cache on more than two cores.
- Mechanism to dynamically decide the number of queue elements that stores the SBPMDTA values.

REFERENCES

1. Nakeeb, N. and Kucuk, G. Set-Based Dynamic Cache Partitioning On Chip Multiprocessor. *International Symposium on Computing in Science and Engineering,* İstanbul, 1-10, June 2010.

2. Stone, H.,S., Turek, J. and Wolf, J.,L. Optimal Partitioning of Cache Memory. *IEEE Transactions on Computers,* 41(9):1054-1068, September 1992.

3. Chiou, D., Jain, P, Devadas, S. and Rudolph L. Dynamic Cache Partitioning via Columnization. *in Design Automation Conference*, 35-42, 2000.

4. Suh, G. Edwards, Devadas, S. and Rudolph, L. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. *in International Symposium on High-Performance Computer Architecture*-8, 2002.

5. Suh, G. Edwards, Devadas, S. and Rudolph, L. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing,* (28)1, 2004.

6. Kim, S., Chandra, D. and Solihin, Y. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. *in Parallel Architectures and Compilation Techniques - 13,* 2004.

7. Hsu, Lisa R., Reinhardt, S., K., Iyer, R. and Makineni, S. ”Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as Shared Resource”. *in Parallel Architectures and Compilation Techniques -15, 2006*.

8. Settle, A., Connors, D., Gibert, E. and Gonzalez, A. Dynamically Reconfigurable Cache for Multithreaded Processors. *Journal of Embedded Computing,* 1(3-4), 2005.

9. Lin, J., Lu, Q., Ding, X., Zhang, Z. and Sadayappan, P. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. *in International Symposium on High-Performance Computer Architecture,* 2008.

10. Rafique, N., Lim W-K. and Thottethodi, M. Architectural Support for Operating System-Driven CMP Cache Management. *in Parallel Architectures and Compilation Techniques, 2006*.

11. Qureshi, M., K. and Patt, Y., N. 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.

12. Moreto, M., Cazorla F., Ramirez, A. and Valero, M. Dynamic Cache Partitioning Based on the MLP of Cache Misses. *in Parallel Architectures and Compilation Techniques* -16, 2008.

13. Moreto, M., Cazorla F., Ramirez, A. and Valero, M. Explaining Dynamic Cache Speed Ups. *IEEE CAL, 2007*.

14. Sharkey, J.,J., Ponomarev, D. and Ghose Kanad. M-SIM: A Flexible, Multithreaded Architectural Simulation Environmen. *Technical Report CS-TR-05-DP01*. Department of Computer Science. Binghamton University. 2005.

15. SPEC. ‘’*Standard Performance Evaluation Corporation*’’, http://www.spec.org/benchmarks.html, (Retrieved: 10.11.2015).