EVALUATION OF LAST LEVEL SET-BASED CACHE PARTITIONING TECHNIQUES IN TERMS OF PERFORMANCE, POWER AND FAIRNESS

by

İsa Ahmet Güney

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

Yeditepe University
2020

EVALUATION OF LAST LEVEL SET-BASED CACHE PARTITIONING TECHNIQUES IN TERMS OF PERFORMANCE, POWER AND FAIRNESS

APPROVED BY:

Prof. Dr. Gürhan KÜÇÜK ....
(Thesis Supervisor)
(Yeditepe University)

Prof. Dr. Sezer GÖREN UĞURDAĞ ....
(Yeditepe University)

Prof. Dr. Fatih UĞURDAĞ ....
(Özyeğin University)

Assist. Prof. Dr. Onur DEMİR ....
(Yeditepe University)

Assist. Prof. Dr. Erdinç ÖZTÜRK ....
(Sabancı University)

DATE OF APPROVAL: ..../..../2020

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Assoc. Prof. Gürhan Küçük for his help, guidance and encouragment throughout the research.

I also thank Burak Sezin Ovant and Abdullah Yıldız for their help with my research work.

Finally, I would like to thank my wife and my family for supporting me.

ABSTRACT

EVALUATION OF LAST LEVEL SET-BASED CACHE PARTITIONING TECHNIQUES IN TERMS OF PERFORMANCE, POWER AND FAIRNESS

Last Level Cache is an important resource that is shared among many cores in multi-core processors. In an unmanaged system, this resource is allocated on demand, which may cause performance problems when there is not sufficient cache space for all running applications. In order to achieve higher system performance, smart partitioning mechanisms must be utilized. This thesis describes a scalable, fine-grain and high-performance set-based partitioning mechanism, along with a tailor-made allocation policy. Evaluations yield that the proposed mechanism can provide 5 percent improvement on average, in terms of throughput and fairness, compared to Vantage partitioner, state-of-the-art in the literature. Additionally, set-based partitioning is investigated as a potential energy-saving technique. Average energy consumed per instruction by the cache is reduced by 28 percent, while losing less than 1 percent throughput on average, with minor modifications in the allocation policy and enforcement scheme of the proposed set-based partitioner.

ÖZET

SET BAZLI SON SEVİYE ÖNBELLEK PAYLAŞTIRMA TEKNİKLERİNİN PERFORMANS, GÜÇ VE ADİLLİK BAKIMINDAN DEĞERLENDİRİLMESİ

Son Seviye Önbellek, çok çekirdekli işlemcilerde birçok çekirdek tarafından paylaşılan önemli bir kaynaktır. Kontrolsüz bir sistemde bu kaynak talebe bağlı olarak dağıtılmaktadır. Bu durum, çalışmakta olan tüm uygulamalar için yeterli önbellek alanı bulunmadığında performans yönünden sorunlar doğurabilmektedir. Yüksek sistem performansı elde edebilmek için akıllı paylaştırıcı mekanizmalara ihtiyaç duyulmaktadır. Bu tezde ölçeklenebilir, hassas ve yüksek performanslı, set bazlı bir paylaştırma mekanizması ve ona özel hazırlanmış bir kaynak dağıtma algoritması tarif edilmektedir. Değerlendirmelere göre önerilen mekanizmanın, literatürden alınan Vantage paylaştırma mekanizmasından ortalama yüzde 5 daha yüksek komut tamamlama oranı ve adillik sağladığı görülmüştür. Ek olarak, set bazlı paylaştırma potansiyel bir enerji tasarruf yöntemi olarak incelenmiştir. Önerilen set bazlı kaynak dağıtma algoritması ve paylaştırma mekanizmasına yapılan ufak düzenlemelerle ortalama komut tamamlama oranı kaybı yüzde 1'in altında iken önbellek tarafından komut başına harcanan enerjide ortalama yüzde 28 tasarruf sağlanmıştır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF TABLES x
LIST OF SYMBOLS/ABBREVIATIONS xi
1. INTRODUCTION 1
2. SET PARTITIONING 8
2.1. SET PARTITIONING 8
2.1.1. Set Index Computation 9
2.1.2. Double Access Mechanism 10
2.1.3. Ensuring Correct Computation 12
2.1.4. Fast Set Redirection 15
2.2. SET CLASSIFIER 16
3. SET PARTITIONING AS AN ENERGY-SAVING TECHNIQUE 22
3.1. MOTIVATION 22
3.1.1. Cache Decay 26
3.1.2. Drowsy Cache 32
3.2. ADDITIONAL ENERGY COSTS OF SET PARTITIONING 37
3.2.1. Fast Set Redirection Hardware 37
3.2.2. Other Energy Costs 39
3.3. A RULE-BASED APPROACH FOR SET PARTITIONING 40
4. TESTS AND RESULTS 43
4.1. SET PARTITIONING 43
4.1.1. Experimental Methodology 43
4.1.2. Results and Discussion 45
4.2. ENERGY CLASSIFIER 55
4.2.1. Experimental Methodology 56

4.2.2. Results and Discussion 57

5. RELATED WORKS 63

5.1. SIMILAR WORK 63

5.1.1. DR-SNUCA 64

5.1.2. Fair and Adaptive Online Set-based Cache Partitioning 65

5.1.3. Jigsaw 66

5.2. VANTAGE 67

6. CONCLUSIONS AND FUTURE WORK 69

7. LIMITATIONS 72

REFERENCES 74

LIST OF FIGURES

Figure 1.1. Example cache organizations for various partitioning approaches. (a) An example organization for a way-based partitioning. (b) An example organization for a line-based partitioning. (c) An example organization for a set-based partitioning. 5

Figure 2.1. Example set-index computations for traditional and Set Partitioning caches. (a) An example set-index computation in a traditional cache. (b) An example set-index computation in Set Partitioning cache. Partition 0 is allocated 200 sets and Partition 1 is allocated 300 sets. The incoming line is inserted into Partition 1. (c) An example set-index computation in Set Partitioning cache. Partition 0 is allocated 300 sets and Partition 1 is allocated 200 sets. The incoming line is inserted into Partition 1........... 11

Figure 2.2. Anexample case where retaining a clean cache line may cause errors. 13

Figure 2.3. Logic design of Fast Set Redirection. 17

Figure 3.1. The average rate of energy spent by various configurations. 28

Figure 3.2. Average throughput loss of various configurations. 29

Figure 3.3. The average efficiency of various configurations. 30

Figure 3.4. The average rate of energy spent by various configurations. 34

Figure 3.5. The average throughput loss of various configurations. 35

Figure 3.6. The average efficiency of various configurations. 37

Figure 4.1. Throughput and fairness gain of Set Classifier. 47

Figure 4.2. Throughput and fairness gain of Set Partitioning with DA over without DA...48

Figure 4.3. Throughput and fairness gain of Set Partitioning with FSR over without FSR.50

Figure 4.4. Entropy gain for FSR compared to modulo operator. 51

Figure 4.5. Additional conflicts for FSR compared to modulo operator. 51

Figure 4.6. Average entropy gain for FSR compared to modulo operator. 52

Figure 4.7. Average increase in cache misses for FSR compared to modulo operator. 53

Figure 4.8. Throughput and fairness gains of Set Partitioning over Vantage. 54

Figure 4.9. The average energy spent by Energy Classifier with various LIMIT values. ...58

Figure 4.10. The average energy spent per instruction by Energy Classifier. 60

Figure 4.11. The average throughput loss caused by Energy Classifier. 61

Figure 4.12. The average energy efficiency gain of Energy Classifier. 62

LIST OF TABLES

Table 3.1. Simulated system specifications 25

Table 3.2. Power/energy requirements of the simulated L2 caches 25

Table 3.3. Cache Decay parameters 27

Table 4.1. Workloads 44

Table 4.2. Processor configurations 44

Table 4.3. Parameters of the partitioning algorithm 45

Table 4.4. Specifications of the simulation environment 56

Table 4.5. List of workloads 57

LIST OF SYMBOLS/ABBREVIATIONS

CPI	Cycles per instruction
CurrentTS	Current timestamp
D	Decay
DA	Double access
DRate	Decay rate
$DThresh_{low}$	Decay rate lower threshold
$DThresh_{high}$	Decay rate upper threshold
DNUCA	Dynamic non-uniform cache architecture
E	Energy
E/Inst	Energy per instruction
FRFCFS	First ready first come first serve
FSR	Fast set redirection
GB	Gigabyte
GHz	Gigahertz
IPC	Instructions per cycle
K	Kilo
KB	Kilobyte
L1	Level 1
L2	Level 2
LLC	Last level cache
LRU	Least recently used
M	Miss
MRate	Miss rate
MRU	Most recently rsed
$MThresh_{low}$	Miss rate lower threshold
$MThresh_{high}$	Miss rate upper threshold
mW	Miliwatt
ns	Nanosecond
NUCA	Non-uniform cache architecture
OoO	Out of order

PIPP	Promotion/insertion pseudo-partitioning
pJ	Picojoule
ROB	Reorder buffer
SC	Set classifier
SetpointTS	Setpoint timestamp
SNUCA	Static non-uniform cache architeture
STB	Share translation buffer
T	Traffic
TC	Threadclass
TRate	Traffic rate
$TTresh_{low}$	Traffic rate lower threshold
$TTresh_{high}$	Traffic rate upper threshold
TV	Threadvalue
U	Unit
UCA	Uniform cache architecture
UMON	Utility monitor
uW	Microwatt

1. INTRODUCTION

The importance of cache performance has been ever-increasing for the past few decades. Processor performance improved much faster than memory structure performance in the past few decades, causing the processing cores to spend more time idly, waiting for the data to be fetched from the main memory. Caches help fight this problem, called the Memory Wall, by storing critical data in much smaller but faster storage area which is also physically close to the processor. As the performance gap between processing cores and memory structures widened, the impact of cache performance became more significant. The increase in the number of processing cores present in a single chip also plays an important role in the increased pressure on cache structures.

The most common cache configurations for multi-core processors include private L1 and L2 caches along with a shared Last Level Cache (LLC). Sharing a cache level among multiple cores introduces a number of problems: as with any case where the resources are limited but the demand is virtually infinite, some sort of regulation is required to achieve near-optimal cache performance.

Several problems may occur in an unregulated shared cache. Some applications may *monopolize* the cache space by accessing the cache more frequently than some others, forcing their data to be evicted. Even worse, the monopolizing application may have stream-like properties where the data inserted to the cache are not reused for such long times that recently inserted lines become practically useless. This kind of behavior is called *thrashing* and is harmful to the system performance in addition to the fairness degradation caused by monopolization.

A lack of regulation in a shared cache may also cause under-utilization. A good policy may detect when and how much an application will benefit from additional cache space, and decide to take away cache space from one application to another if the benefits outweigh the losses. Moreover, applications tend to experience significant improvements in their cache performance once they obtain sufficient cache space to fit their working sets in. Policies that can guarantee that applications will get enough cache space can exploit such performance gains, or they can prevent resource waste when applications do not show any interest in extra cache space.

There are three ways to manage a shared cache: 1) *sharing*, 2) *partitioning* and 3) *partition-sharing*. In sharing, all applications race to steal cache space, causing the problems discussed above. In partitioning, each application is allocated a certain cache space. Finally, in partition-sharing, applications may be divided into groups, where each group is allocated cache space as partitioning, and applications in the same group share that cache space among themselves as in sharing. Sharing is a special case of partition-sharing where all applications belong to the same group. Brock et al. show that partitioning is always better than partition-sharing [1], which motivates researchers to investigate cache partitioning mechanisms to obtain near-optimal cache performance.

Cache partitioning has two major components: the allocation policy and the enforcement mechanism. The allocation policy is responsible for deciding what portion of a cache each partition should receive. The enforcement mechanisms' duty, on the other hand, is to make this allocation decision happen.

Enforcement mechanisms can be investigated in two categories: implicit and explicit mechanisms. Implicit ones are the mechanisms that try to guide the system towards the desired allocation by modifying the cache eviction/insertion policy. For example, Promotion/Insertion Pseudo-Partitioning (PIPP) inserts incoming lines into recency positions different that the most recently used position, according to the owning partition's allocation [2].

On the other hand, explicit mechanisms enforce the allocation by limiting eviction candidates. For example, in way partitioning, a partition can only evict one of its own lines if that partition has reached its target size in that set [3]. Similarly, a partition can only evict a line belonging to another partition if it is below its target size.

There are some important features that enforcement schemes should support to achieve high-performance levels. With the ever-increasing number of cores in a processor, scalability becomes an important feature that every enforcement scheme should consider. Recently, Intel and AMD released their 32- and 64-core architectures, Xeon Phi and Ryzen Threadripper.

Considering that ideally, partitioning should be used rather than partition-sharing, a partitioning mechanism should support at least as many partitions as the number of cores in

the system, and perhaps more, in order to support separate partitions for individual threads or orthogonal techniques such as Talus [4]. For example, one of the most well-known techniques, way partitioning can support only as many partitions as the associativity of the cache. However, since increasing associativity beyond a certain level starts hurting performance due to the increase in access latency, there exists a virtual limit on associativity, which also limits the scalability of any partitioning mechanism which solely relies on cache ways.

In order to be truly scalable, a mechanism should use a highly available base unit. There are usually only a handful of cache ways in a cache, but thousands of cache lines. Therefore, partitioning a cache in terms of cache lines would be a much finer-grain way of partitioning as well as being highly scalable. Indeed, mechanisms such as Vantage partition the cache in terms of cache lines [5]. In Vantage, a partition may have a different number of lines in different sets, as long as the total number of lines the partition has is equal (or close) to its target size. However, this approach also has its problems, such as isolation and preservation of associativity. In Vantage, an application may steal lines from another application even when the second application is not over its target size.

Providing isolation is an important aspect of cache partitioning. Violating isolation may cause important lines to be evicted by other applications, causing harmful behavior. This is the harmful behavior cache partitioning techniques are trying to prevent in the first place. In addition to its impact on performance, lack of isolation also harms fairness.

A partitioning mechanism should also preserve the associativity. The ultimate purpose of partitioning mechanisms is to improve system performance. However, hurting effective associativity may have an even greater impact on performance. This problem becomes even more significant in configurations where a high number of partitions are required. For example, in an 8-way cache with 8 partitions, way partitioning will have to allocate only one way per partition, which is a solution in which partitions are effectively using a direct-mapped cache. In line-based mechanisms such as Vantage, this problem still exists even though it is less severe. The number of eviction candidates in a given set will be determined by which partitions are existent in the set, and whether they have achieved their target sizes or not. Even if the cache reaches a stable state where the actual sizes of partitions are exactly

equal to their target sizes, the number of eviction candidates available will be equal to the number of lines the inserting partition has in the respective set.

In summary, scalability (fine-grain partitioning, support for a high number of partitions), isolation (protecting partitions from each other) and preservation of associativity (keeping the number of eviction candidates high) are three pillars of a good cache partitioning mechanism. It is these factors that motivate this study: partitioning the cache in terms of sets supports these key features. It is fine-grain because a cache set is a fairly small unit of storage, and one can give/take away a single set to/from a partition. It is scalable since a very large number of cache sets are usually present in a last-level cache (hundreds/thousands of sets). It can also provide isolation by preventing partitions from evicting cache lines from other partitions. Lastly, it can preserve associativity since any given set will be completely dedicated to a single partition during an allocation decision. Figure 1.1. shows example organizations for way-based, line-based and set-based partitioning mechanisms.

Naturally, there are some challenges along the way to develop a set-based partitioning mechanism. The two most prominent challenges are the problem of directing accesses to the sets owned by the partition, and the problem of preventing cache lines to be lost after a re-allocation. Along with these problems, there also exist some sub-mechanisms to investigate which may further improve the performance of set-based partitioning.

Today, the most growing concern in processor development is energy consumption. With the increase in processor use in household devices and the increase in both the number and size of data centers, the importance of energy efficiency became much more important than ever. A recent study predicts that if no precaution is taken, energy consumed by information and communication technologies may amount to more than 20 percent of global energy consumption by 2030 [6].

Cache partitioning mechanisms proposed in the literature usually focus on performance rather than energy efficiency. However, cache memory is also a good candidate for saving energy. There exist some line-based energy saving techniques which aim to save energy by powering down unused cache lines, or putting them into a low-power mode [7,8]. Still, a smart energy-oriented cache allocation policy may provide even better energy savings by orchestrating cache management in the long run.

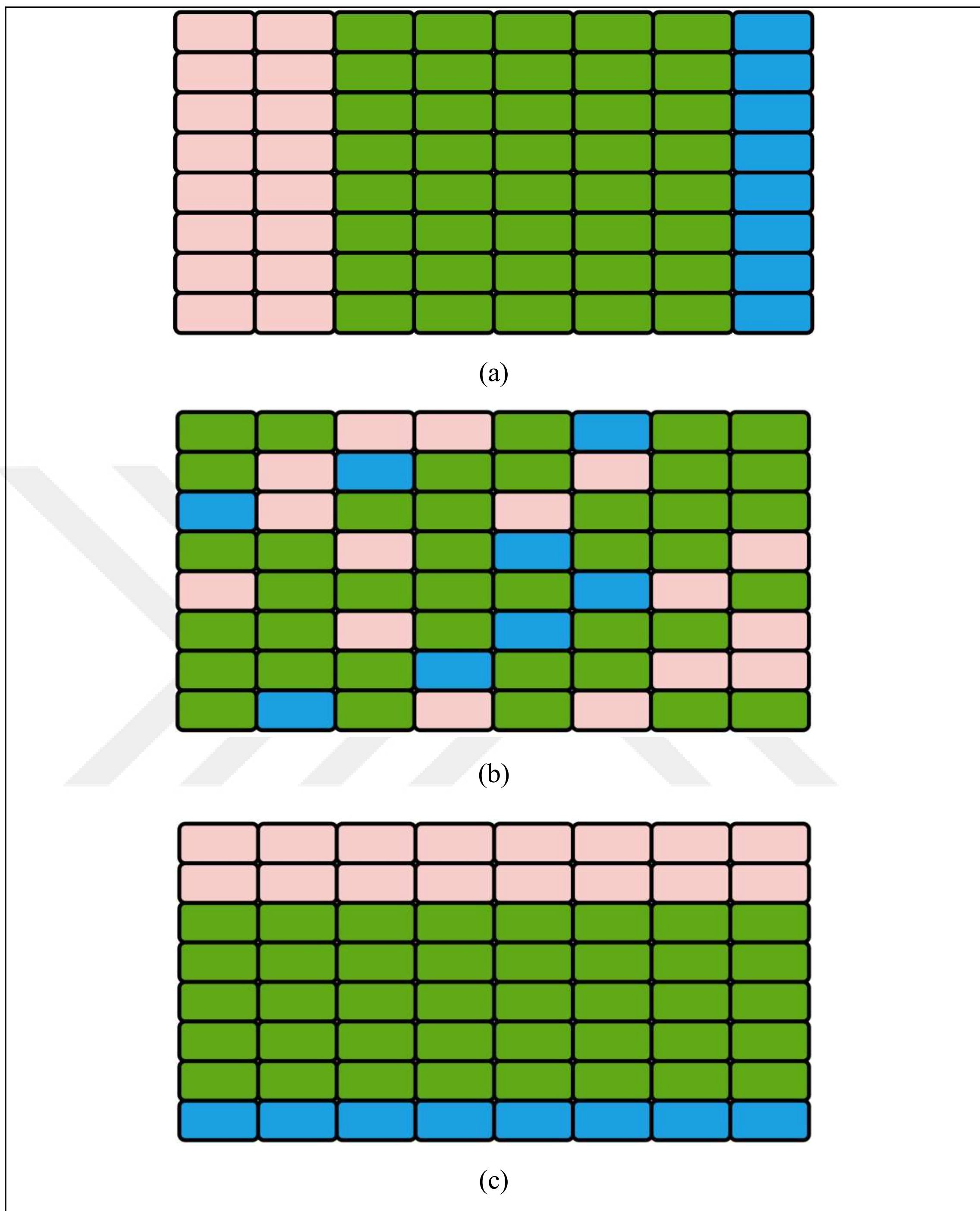


Figure 1.1. Example cache organizations for various partitioning approaches. (a) An example organization for a way-based partitioning. (b) An example organization for a line-based partitioning. (c) An example organization for a set-based partitioning.

We propose a set-based cache partitioning with the primary focus of improving the performance and fairness of the systems. Our contribution in this research are listed as follows:

- Introduction of a set-based cache partitioning mechanism:
    - An access redirection circuitry that can redirect accesses to any range of sets is proposed. The effects of utilizing this circuitry in terms of performance, fairness and energy are analyzed.
    - The effects of invalidating cache lines periodically due to the change in allocation decisions are analyzed.
    - A double-access mechanism that improves performance by allowing partitions to access lines based on previous allocation decisions is introduced. The contributions of this mechanism in terms of performance and fairness are analyzed.
- Introduction of a new allocation policy:
    - Scalable allocation policy that has a linear time complexity which is only dependent on the number of partitions supported.
    - Classifies access streams according to their cache behavior.
    - Designed specifically with set-based partitioning in mind.
- Comparison of the introduced partitioning scheme with a line-based scheme called Vantage from literature in a many-core configuration.
- Modifying the partitioning scheme to decrease energy consumption/improve energy efficiency:
    - A preliminary analysis which treats set-based partitioning as a special case of powering down portions of cache, and compares it with other line-based energy-saving techniques (Cache Decay and Drowsy Cache).
    - Analysis of the impact of set-based partitioning and its sub-mechanisms on energy consumption.
    - Introduction of a proof-of-concept energy-oriented allocation policy, which is very similar to the original allocation policy, but can also provide a reduction in energy consumption.

The rest of the thesis is structured as follows: Chapter 2. describes the proposed partitioning scheme and allocation policy. Chapter 3. analyzes set-based partitioning as an energy-saving

technique and proposes an example allocation policy. Chapter 4. presents simulation results and discussion. Chapter 5. summarizes background work in the field and discussed the differences between this research and similar work from the literature. Chapter 6. concludes by summarizing the research and discussing the importance of this work, along with possible future work. Chapter 7. discusses the limitations of this research.

2. SET PARTITIONING

The organization of this chapter is as follows: Chapter 2.1. explains the enforcement scheme of the partitioner by describing the proposed set-index computation, the Double Access mechanism that initiates a secondary access after a cache miss, the forced eviction process which prevents the cache from entering a faulty state and the proposed circuitry for the new set-index computation. Chapter 2.2. presents the proposed allocation policy that works as a counterpart to the enforcement scheme.

2.1. SET PARTITIONING

Set Partitioning is an enforcement mechanism, which partitions the cache by allocating blocks of sets to partitions. In this mechanism, associativity available to a partition is not hindered by other partitions. Partitions work with cache regions, which have less number of sets but are completely dedicated to themselves.

At a glance, advantages of Set Partitioning are better scalability, granularity and performance. In Set Partitioning, the smallest unit which can be allocated to a partition is a set. Caches usually consist of hundreds or thousands of sets, which makes Set Partitioning a fine-grained mechanism. This also makes Set Partitioning scalable since there are a large number of sets available for partitioning; thus, Set Partitioning can support a large number of partitions.

Partitioning the cache in terms of sets allows Set Partitioning to preserve associativity provided by the hardware. For example, way partitioning halves associativity for each partition in a two-partition configuration whereas Set Partitioning does not degrade associativity. Besides, associativity stays constant as the number of partitions increase in Set Partitioning.

Set Partitioning also raises some issues as cache coherency, cache flush effect and the latency of modulo operator. These issues and our proposed solutions are discussed in the following subsections.

2.1.1. Set Index Computation

In traditional caches, the set index of a memory address is determined by applying a bitmask to the memory address after eliminating offset bits. This is equivalent to computing the modulo as shown in Equation 2.1. However, since each partition owns a reduced number of sets in Set Partitioning, the set index is computed using the number of sets allocated to a partition as shown in Equation 2.2.

$$set\ index \quad (memory\ address)\ mod\ (number\ of\ cache\ sets) \tag{2.1}$$

$$set\ index = beginn \quad set_i + (memory\ address)\ mod\ (number\ of\ cache\ sets_i) \tag{2.2}$$

The beginning set of a partition starts at the end of the previous partition's dedicated cache region and can be computed iteratively using Equation 2.3.

$$beginning\ set_i = \begin{cases} 0, & if\ i = 0 \\ beginning\ set_{i-1} + size_{i-1}, & otherwise \end{cases} \tag{2.3}$$

An example organization of Set Partitioning with two cores and a cache of 500 sets is as follows: *core 0* and *core 1* are given 200 and 300 sets, respectively. When *core 1* accesses the address 1550, the set index is computed as (200 + 1550 mod 300) and is inserted into set 250. However, after a repartitioning phase, which gives 300 sets to *core 0* and 200 sets to *core 1*, same address will be mapped to set 450, computed as (300 + 1550 mod 200). This example is depicted in Figure 2.1.

In traditional caches, lines with different set bits are directed into different sets, and lines that have identical set bits are distinguished by their tag bits. However, when Set Partitioning is utilized, cores may have less number of sets than there is in the cache, causing lines which have identical tags but different set bits being mapped to the same set after repartitioning. To help the cache distinguish different addresses that are mapped to the same set, tag bits are extended to include set bits in Set Partitioning. The additional area requirement due to this extension is somewhat negligible. For example, in a 2048 set cache with 64-byte line size in a 32-bit address space, each cache line requires 537 bits (512 bits for data, 15 bits for the tag, 8 bits for timestamp, 1 bit for a valid flag and 1 bit for a dirty flag). With Set Partitioning, this value increases to 549, since 11 bits are added to the tag field of each cache

line to include set bits and a one-bit counter is utilized for invalidating stale data (discussed in Chapter 2.1.3.). For this configuration, the inclusion of set bits causes roughly a 2 percent increase on the entire cache space.

2.1.2. Double Access Mechanism

Changing the mapping function is equivalent to an immediate cache flush in terms of throughput. Lines may remain in the cache, but most of them, if not all, become effectively inaccessible. Chapter 2.1.1. presents an example where a cache line which actually is in the cache becomes inaccessible due to a change in the mapping function. In order to mitigate the throughput loss caused by these cache flushes, a new mechanism called Double Access (DA) is proposed.

In DA, partitions remember their size and beginning set for the previous allocation in addition to the current one. This allows partitions to be able to locate their valuable cache lines inserted in earlier periods. When a cache access occurs, a partition initially searches a cache line within a set indexed by the current mapping function. If the line is valid and if there is a tag match, it is called a *primary hit*, otherwise a *primary miss*. In the case of a primary miss, the partition searches the requested line within a set computed by the mapping function of the previous period. If the line is valid and if there is a tag match at the end of this second search, it is called a *secondary hit*, otherwise a *secondary miss*. Primary and secondary accesses are carried out sequentially; therefore, access latency of a primary hit is equal to the latency of the cache, whereas the access latency of a secondary hit is equal to twice the latency of the cache. In case of a secondary miss, data is retrieved from a lower-level memory structure as usual.

Secondary accesses allow partitions to address cache lines from other partitions. Partitions are free to evict any line from their dedicated space whenever it is necessary, but there is no need to prevent others from accessing these lines as long as they stay healthy in the cache, especially if others can benefit from them. Indeed, results discussed in Chapter 4.1.2.2. show that there is a possible throughput gain by allowing this via the DA mechanism.

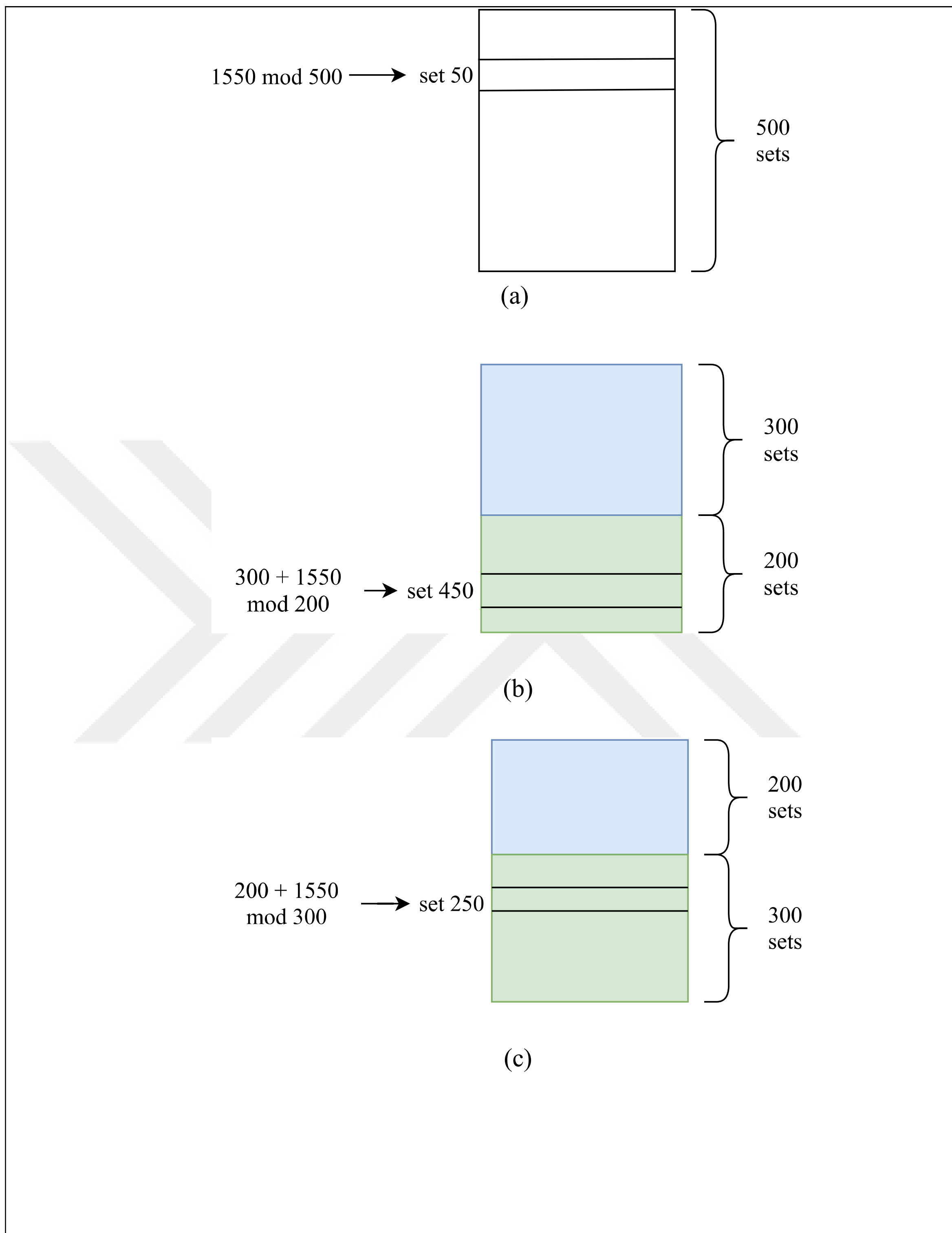


Figure 2.1. Example set-index computations for traditional and Set Partitioning caches. (a) An example set-index computation in a traditional cache. (b) An example set-index computation in Set Partitioning cache. Partition 0 is allocated 200 sets and Partition 1 is allocated 300 sets. The incoming line is inserted into Partition 1. (c) An example set-index computation in Set Partitioning cache. Partition 0 is allocated 300 sets and Partition 1 is allocated 200 sets. The incoming line is inserted into Partition 1.

In DA, to prevent partitions from evicting lines from other partitions and violating allocation decisions, the secondary access only allows accessing cache lines without updating their timestamps, and all insertions are applied to a partition's own space (i.e. using the most up-to-date mapping function). Another interesting design choice might be to update timestamp information of cache lines that give a *secondary hit*. In such a case, the new owner of those cache lines may get a really hard time evicting them, since they can be instantly moved to the Most Recently Used (MRU) position in a cache set, with each secondary hit. Currently, giving secondary access as a best-effort service without impeding the performance of the new owner of a partition is utilized.

Another design choice to be considered when applying the DA mechanism is to decide whether a line should be relocated to its primary partition upon a secondary hit or not. Here, relocating a frequently used line halves the access time, whereas relocating a rarely used line initiates unnecessary traffic and possibly evicts a useful line from the primary partition. A more ambitious option would be to relocate all lines upon a change in the mapping function. As discussed in Chapter 2.1.3., the proposed scheme requires bulk invalidations to be carried out at the end of each repartitioning period. This approach can eliminate the DA mechanism altogether and minimize the number of writebacks. However, this also requires further analysis of design space and its impact on performance. The current scheme with no relocations costs much less both in terms of cache traffic and hardware resources. Cache lines that are accessed in foreign partitions are gradually evicted by the new owners of partitions, and frequently used lines are inserted into primary partitions upon a cache miss.

2.1.3. Ensuring Correct Computation

When a partition's beginning set or size changes, its mapping function for computing set indices also changes. This may cause lines which are inserted two periods ago to be inaccessible (unless the mapping function happens to be the same with one of the last two periods). If not managed properly, such lines will become "lost", i.e. will still be in the cache but become inaccessible.

Lost lines cause two types of problems which may cause the program to enter a faulty state. If the lost line is a dirty line, the cache will be unable to locate the requested line and bring the line with an outdated value from a lower-level memory structure. The other case is when

the lost line is clean: a clean line may be retained in the cache long enough so that it becomes accessible again after several repartition periods, but data in that line may be outdated.

An example of the latter case is shown in Figure 2.2. Let A be a variable of interest in an application. The application inserts the related line let us call line L, into the cache at period *t*. Due to change in mapping functions, L becomes inaccessible at period *t+2*. The application again accesses A at *t+3*. Since L is inaccessible, another line (L') is inserted into the cache. At *t+4*, the value of A changes. The value stored in L' is also updated, but since L is inaccessible at this point, the value in L becomes outdated. At *t+5*, L' becomes inaccessible. At *t+6*, L is still alive and becomes accessible again due to a change in the mapping function. At *t+7*, the application accesses A again and uses the outdated data stored in L.

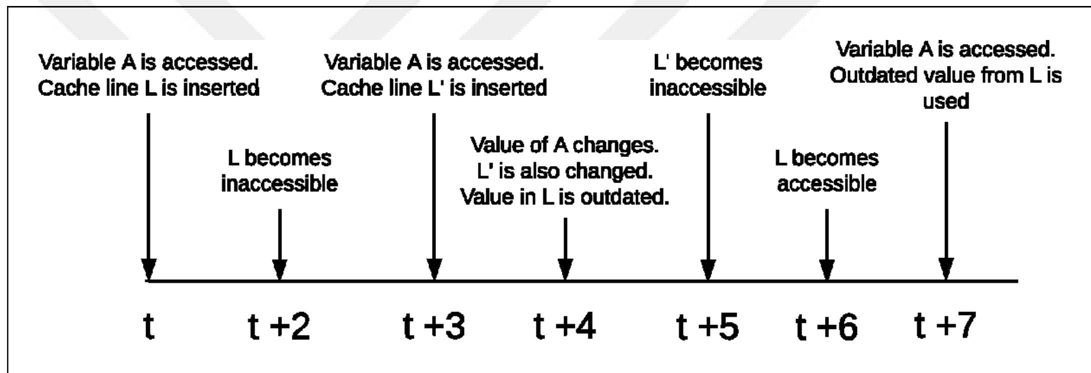


Figure 2.2. Anexample case where retaining a clean cache line may cause errors.

Both problems arise from lost cache lines. Therefore, to prevent such problems, lost cache lines must be avoided at all costs. This can be implemented by using an extra one-bit counter per line, called the *stale bit*. This bit represents whether a cache line will become lost in the next repartitioning period or not. If a cache line is just inserted or is accessed by a primary access, it means that it will also be accessible during the next period. Therefore, when a line is inserted or accessed via a primary access, the stale bit of that line is reset.

At the end of each period, all lines which will be lost during the next period must be evicted. Similar to a traditional eviction, if the line to be evicted is clean, simply resetting the valid bit is sufficient to evict that line. On the other hand, if the line is dirty, the contents of that line must be sent to the lower level unit in the memory hierarchy. Algorithm 2.1. and Algorithm 2.2. show how cache lines are treated at each access and the end of each repartitioning period.

Algorithm 2.1. Access cache line

```
if primary access then
    line.stale_bit ← 0
end if
```

Algorithm 2.2. Flush stale lines

```
for every cache line l do
    if l.stale_bit == 0 then
        l.stale_bit ← 1
    else
        if l.dirty == 1 then
            Send data of l to the lower level
        end if
        Invalidate l
    end if
end for
```

There exist some solutions which could significantly reduce the time spent for invalidating dirty cache lines. For example, relocating lines on secondary hits would move some of the dirty stale bits to a primary partition, rendering them *not stale*, hence eliminating the need for writing these lines back to memory. Similarly, the eviction process could be handled gradually towards the end of repartitioning periods and using write-through policy during this interval would prevent creating new dirty lines. Finally, an extreme approach would be to use the write-through cache policy from the beginning to the end.

However, these solutions are either not practical, or do not guarantee any performance gains. Utilizing write-through policy would greatly increase memory bandwidth utilization and power consumption. Relocating lines on secondary hits could help, but there is no guarantee on what percentage of dirty stale lines would be relocated. Besides, as discussed in Chapter 2.1.2., relocating lines may hurt performance and requires further analysis.

Here, a bulk invalidation/eviction process after each repartitioning period is proposed. During this process, all cache lines are sequentially checked, and clean stale lines are invalidated, and dirty stale lines are written back to the main memory. The process ends when all lines are checked and all write-back operations are completed. All cache accesses are blocked during this period. The impact of this invalidation/eviction process on performance is discussed in Chapter 4.1.2.4. Support for bulk invalidation of cache lines have been proposed in other work in the literature as well [9,10].

2.1.4. Fast Set Redirection

The modulo operator is an ideal operator for directing incoming cache accesses towards cache sets if addresses are assumed to be uniformly distributed. Since the computation of the set index is on the critical path of cache access, the number of sets in caches are always selected in powers of two to allow fast bit extraction instead of executing a costly modulo operation. However, in Set Partitioning, partitions may have sizes that are not powers of two. The modulo operator is significantly slower than the bit extraction and it is an unfeasible option for fast set index computation. Considering that cache access latency has a big impact on the processor throughput, the mechanism which replaces modulo operator should be as simple and as fast as possible.

As speed is of utmost importance, utilizing bitmasks in order to harness their speed is proposed. Two simple approaches exist which use bitmasks with non-power of two sizes: rounding a given size *up* or *down* to a power of two.

If partition sizes are rounded down, some sets would never be indexed and, therefore, never would be accessed. This approach would require additional mechanisms for deciding which accesses would be directed to that extra space and how accesses would be indexed within that region. On the other hand, if partition sizes are rounded up, some accesses would be indexed to sets which are beyond the extent of a partition and would have to be redirected inside partition boundaries.

A simple and fast method called Fast Set Redirection (FSR) is proposed for computing set indexes by rounding partition sizes up to a power of two as shown in Algorithm 2.3.

Algorithm 2.3. Fast Set Redirection

```
set index = (address) mod (rounded up partition size)
if set index ≥ partition size then
        set index = set index – partition size
end if
set index = set index + beginning set of partition
```

The proposed set index computation method is not ideal and does not uniformly distribute accesses. Therefore, some sets in a partition are used less frequently than others. The effect of this suboptimal distribution on throughput is examined in Chapter 4.1.2.3.

FSR computation requires one bitmask, one comparison, one subtraction and one addition operations. However, the computation can be implemented in a much simpler way. First, the comparison can be ignored since comparing two operands is carried out by applying a subtraction and checking the sign bit. Therefore, FSR can apply subtraction operation immediately and use this value if the sign bit is appropriate. The addition and subtraction operations can be handled in a single move with the 3-in-1 structures described by Vassiliadis et al. [11]. Overall, FSR computation can be carried out with a bitmask operation and parallel 3-in-1 and 2-in-1 operations. Logic design for FSR is presented in Figure 2.3. Given that bit masking and multiplexing are fast operations, we assume that the critical path of FSR (bit masking, 3-in-1 addition and bit selection) does not necessarily increase cycle time.

2.2. SET CLASSIFIER

Cache partitioning mechanisms consist of two major components: an allocation policy, which decides the amount of cache space each core can receive, and an enforcement policy, which realizes these decisions as faithfully as possible. When these policies work in harmony, near-optimal performance gains can become attainable. Since Set Partitioning is a set-based enforcement policy, we also need a complementary set-based allocation policy. Here, we propose a classification-based policy, which is similar to previous work that solely

based on way partitioning [12]. To make allocation decisions about reference streams, Set Classifier classifies them into four predefined classes: Null (threads which are not interested in the cache), Very Harmful (threads which have destructive cache behavior), Harmful (threads which are mildly destructive, but also benefit from the cache) and Harmless (threads that benefit from the cache greatly). The classification is done periodically in epochs by utilizing run-time statistics. These statistics are the *traffic rate*, *miss rate* and *cache decay rate*.[1]

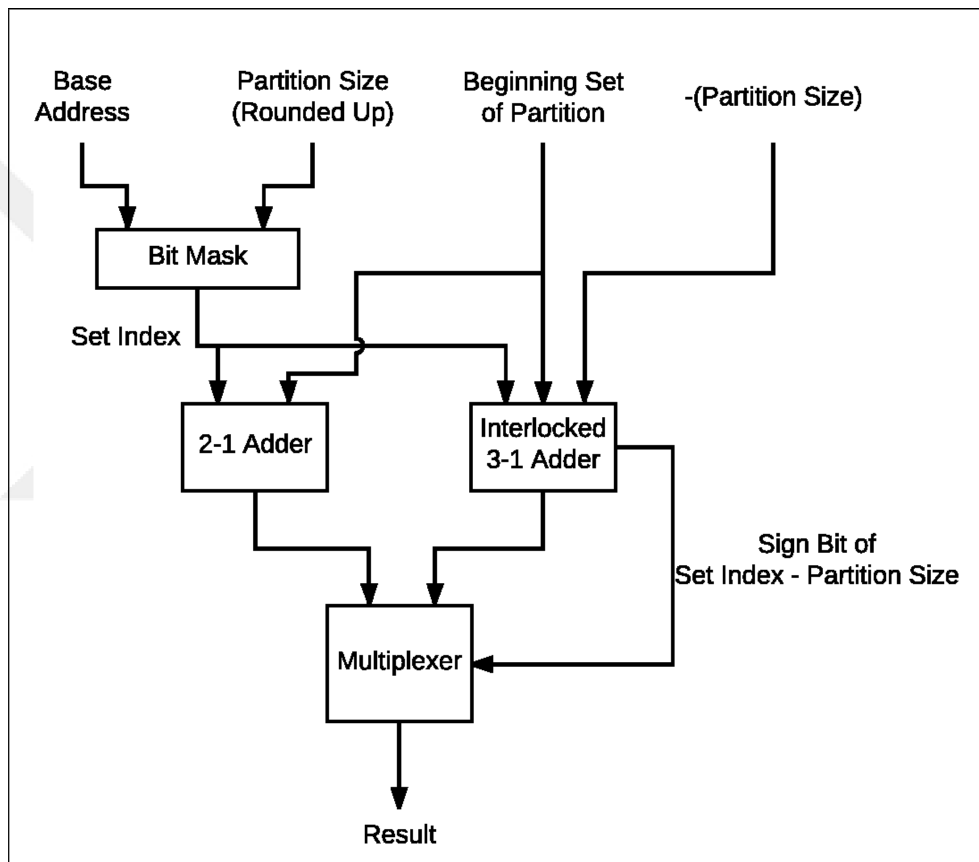


Figure 2.3. Logic design of Fast Set Redirection.

Traffic rates of threads are determined by dividing the number of cache accesses in one epoch to the duration of the epoch (Equation 2.4). Miss rate is determined by dividing the number of cache misses to the number of cache accesses in one epoch (Equation 2.5).

[1] In our work, we assume a fixed one thread per core configuration. Hence, we refer to reference streams as threads in this section. However, the same allocation policy can be easily applied to reference streams with multiple threads by using their cumulative statistics.

Finally, the cache decay rate is determined by dividing the number of cache decays in one epoch to the number of maximum possible cache decays in one epoch (Equation 2.6).

$$TRate_i = \frac{Number\ of\ cache\ accesses_i}{Epo\quad duration} \tag{2.4}$$

$$MRate_i = \frac{Number\ of\ misses_i}{Number\ of\ cache\ accesses_i} \tag{2.5}$$

$$DRate_i = \frac{Number\ of\ cache\ decays_i}{Number\ of\ maximum\ possible\ cache\ decays_i} \tag{2.6}$$

Lookahead allocation policy exploits the stack property of LRU to determine the exact contribution of each allocated way in terms of cache hits that are gained [3]. However, the set-based approach lacks such property and observing the contribution of extra sets on cache hits gained for a particular thread is not as simple as it is done in way partitioning mechanisms. Here, the use of cache decay statistics to gain similar information is proposed. Cache decay is a metric used to determine whether a given cache set (or line) has been idle for a certain amount of time and it is mainly used for reducing the energy consumption of caches [7].

In order to track how long each set has been idle for, per-set decay counters are implemented. Since updating these counters on a per-cycle basis would cause higher complexity and energy dissipation, decay counters are usually implemented in a more coarse-grain fashion. In this research's setup, 3-bit counters per set are implemented. Whenever a set is accessed its decay counter is reset to a *decay reset value*. All counters are decremented by one at the end of a smaller period called *decay period*. In this scheme, a set with a decay value of zero is marked as decayed and the set preserves its decay status until an access occurs to that set. Maximum possible cache decay represents the case where all sets of a thread are decayed throughout the epoch. This value is calculated using Equation 2.7.

$$Maximum\ possible\ decay_i = Number\ of\ sets_i \cdot \frac{Epoch\ duration}{Decay\ period\ length} \tag{2.7}$$

In order to keep the future calculations simple, the statistics collected are classified into 0, 1 and 2 by comparing these values with low and high thresholds. Traffic (T), miss (M) and decay (D) classes of a thread are determined as shown in Equation 2.8, 2.9, and 2.10.

$$T = \begin{cases} 0, & TRate < TThresh_{low} \\ 1, & TThresh_{low} \leq TRate < TThresh_{high} \\ 2, & TThresh_{high} \leq TRate \end{cases} \tag{2.8}$$

$$M = \begin{cases} 0, & MRate < MThresh_{low} \\ 1, & MThresh_{low} \leq MRate < MThresh_{high} \\ 2, & MThresh_{high} \leq MRate \end{cases} \tag{2.9}$$

$$D = \begin{cases} 0, & DRate < DThresh_{low} \\ 1, & DThresh_{low} \leq DRate < DThresh_{high} \\ 2, & DThresh_{high} \leq DRate \end{cases} \tag{2.10}$$

The purpose of Set Classifier is to classify threads according to their cache behavior: threads with high miss rate and/or high decay rate should be classified as Harmful or Very Harmful classes, whereas threads with good cache behavior should be classified as Harmless and threads with no interest in the cache should be classified as Null. After T, M and D classes are determined, each thread is assigned ThreadValues (TV) using the formula in Equation 2.11, to determine how bad their cache behavior is.

$$TV = T \cdot (1 + M + 2 \cdot D) \tag{2.11}$$

The traffic class is multiplied by other statistics since the traffic rate of a thread determines how big the impact of the miss rate and the decay rate is. A thread with a high miss rate will be much more destructive if it also has a high traffic rate.

Similarly, it is much worse if a thread has a high decay rate even though it has a high number of cache accesses (i.e. high traffic rate). One may expect traffic rate and decay rate to have a negative correlation, but this may not be the case with a non-uniform access pattern. If a thread has both high traffic and decay rates, it means that the thread is accessing a subset of cache sets allocated to it very frequently, while accessing the rest of its sets rarely. A thread with a high decay rate is a better candidate for giving away cache space compared to a thread with a high miss rate. Therefore, decay classes of threads are multiplied by 2 to punish threads with high decay rates.

If a thread has a low traffic rate, then, miss rate and decay rate of that thread become insignificant for performance. Such threads are classified as Null (i.e. no interest in any cache resource) and their TV becomes 0. However, Null threads and Harmless threads with good cache behavior (M = 0, D = 0) should be distinguished. Hence, a constant value of 1 is added

to TV computation. This way, threads with high traffic but low miss/decay rates will have a different score than threads with no cache interest.

Once the TV of a thread is determined, ThreadClass (TC) computation determines which category the thread should fall into (Equation 2.12). Reflecting the theory behind Equation 2.11, threads with higher TV's are classified into classes with worse cache behavior.

$$TC \quad \begin{cases} Null, & TV = 0 \\ Harmless, & 0 < TV < 4 \\ Harmful, & 4 \leq TV < 6 \\ Very\ Harmful, & 6 \leq TV \end{cases} \tag{2.12}$$

In Set Classifier, threads, which are classified as Null or Very Harmful, are not given any cache resources since Null threads do not utilize the cache and Very Harmful threads do not gain any performance, similar to previous work by Ovant et al. [12]. As a result, the entire LLC is partitioned among Harmless and Harmful threads. Although it is possible to differentiate the amount of cache space allocated to threads of the same class, it would require dividing these classes into additional subclasses. Thus, all threads of the same class are allocated equal cache space to keep the mechanism simple enough to be implemented in hardware. Since Harmless threads are expected to utilize cache resources more efficiently, they are provided with proportionally (twice, in this study) more cache resources than Harmful threads. How much cache resource a thread should receive is calculated according to Equations 2.13, 2.14 and 2.15.

$$Total\ weight = 2\ \cdot Number\ of\ Harmless\ threads + Number\ of\ Harmful\ threads \tag{2.13}$$

$$U = \frac{Number\ of\ sets\ in\ cach}{Total\ weigh} \tag{2.14}$$

$$Allocation = \begin{cases} floor(2\ \cdot U), & if\ thread\ is\ Harmless \\ floor(U), & if\ thread\ is\ Harmful \end{cases} \tag{2.15}$$

Note that the floor function is used in Equation 2.15, and there might be some leftover sets after partitioning. Leftovers are given to arbitrary threads (these sets are always given to the last thread in this study). In case there are no Harmless or Harmful threads, all cache resources are equally partitioned among Very Harmful threads.

The Set Classifier requires a negligible amount of time to carry out necessary computations. Equations presented in this section are presented with theoretical reasons in mind, but these computations can be simplified in practice. For example, considering Equation 2.4 and 2.8 together, Equation 2.4 can be eliminated by multiplying traffic thresholds by the period length of the classifier. In short, each core must execute 9 comparison operations (6 for determining T, M and D classes; 3 for determining TC value), 2 addition operations (for TV computation) and 3 shift operations (2 for TV computation and 1 for allocation computation in Equation 2.15). Additionally, one core requires to execute 1 shift operation (for Total Weight computation), 1 division operation (for computing U), 35 addition operations (1 for Total Weight computation, 32 for computing total number of sets allocated, 1 for computing number of leftover sets and 1 for adding the leftover sets to a core). In total, Set Classifier imposes a latency of 37 addition, 4 shift, 1 division and 9 comparison operations. When using instruction latencies from Intel Skylake architecture, total latency imposed by these computations is under 100 cycles [13]. Furthermore, the complexity of the Set Classifier is not affected by the associativity of the cache and increases linearly as the number of threads increases.

3. SET PARTITIONING AS AN ENERGY-SAVING TECHNIQUE

This chapter investigates the characteristics of Set Partitioning as an energy-saving technique. Chapter 3.1. motivates by describing energy consumption in caches and comparing a simplified version of Set Partitioning to Cache Decay and Drowsy Cache. Chapter 3.2. discusses the additional energy consumption caused by the implementation of Set Partitioning itself. Chapter 3.3. proposes a new energy-saving allocation policy for Set Partitioning by modifying the existing allocation policy.

3.1. MOTIVATION

Although Set Partitioning is essentially a cache partitioning mechanism, it can also be leveraged as an energy-saving technique. Two important features of Set Partitioning are its ability to isolate cores to a certain cache area and to grow/shrink the area dedicated to a core. When combined, these two features can be exploited to reduce the energy consumption of the cache. The main idea is to power off as many cache sets as possible without hurting performance in order to save energy or follow a more aggressive approach to turn off even more cache sets in exchange for performance loss. Energy saving is then reduced to modifying the allocation policy.

Energy consumption on a cache can be examined in two categories: static and dynamic energy consumption, and the total energy spent by a cache can be expressed as a sum of these two consumption types (Equation 3.1). Static energy consists of the energy spent in order to keep the cache powered on (leakage) and dynamic energy spent consists of the energy spent due to switching of transistor states during a cache access. Dynamic energy spent by a cache can be further broken down into energy spent due to cache reads and energy spent due to cache writes (Equation 3.2).

$$E = E_{static} + E_{dynamic} \tag{3.1}$$

$$E_{dynamic} = number\ of\ cache\ reads\ \cdot\ E_{read} + number\ of\ cache\ writes\ \cdot\ E_{write} \tag{3.2}$$

If a partitioning mechanism supports explicit partitioning, static energy consumption of a shared level cache can be expressed as a sum of static energy consumed by all partitions as

shown in Equation 3.3 where S denotes the cache size, E_i denotes the size of ith partition and E_{static_i} denotes the static energy spent by i^{th} partition.

$$E_{static} = \sum_{i=0}^{N} E_{static} \cdot \frac{S_i}{S} = \sum_{i=0}^{N} E_{static_i} \tag{3.3}$$

The same rule applies for dynamic energy consumption as well: dynamic energy spent by a shared level cache can be expressed as a sum of dynamic energy spent by all partitions. Therefore, the total energy spent by a cache can be expressed as a sum of energy spent by all partitions in the cache. The implication is that any reduction of energy in a single partition can be translated into a reduction of energy in a multi-partition environment. This allows us to simplify the investigation of energy-saving techniques for an explicit partitioning mechanism by investigating in a single-core configuration rather than in a multi-core environment.

One of the properties of an explicit partitioning scheme is that partitions can have a variable amount of cache area allocated only to them. Applied in a single-core configuration, this allows only a subset of cache area to be allocated for work and the rest to be powered off in order to save energy. Gated Vdd can be used to power off individual cache lines [14], and a similar approach is used for instruction caches by Powell et al. [15].

Reducing energy consumption usually results in decreasing performance. In this section, we focus only on the energy consumed by the cache and ignore extra energy costs induced by decreased performance, such as extra energy spent by the processor due to the increased duration of execution, or by main memory due to increased traffic. Still, decreased performance will also induce an increase in static energy dissipated by the cache due to increased execution time, and a lower cache hit rate will mean more dynamic energy will be spent due to the increased number of cache insertions. Moreover, saving energy at the expense of too much performance degradation is not desired, and keeping performance loss within bounds is also an important factor.

Energy-saving techniques will be evaluated using the following metrics: throughput produced by the processor (IPC), total energy spent and energy-saving efficiency. While IPC (and the hit rate of the cache) directly measure system performance, efficiency will indicate how much energy was saved at the cost throughput loss. Traditionally, energy-delay product or energy-delay^2 product is used to measure energy efficiency. However, these two metrics

are not suitable for measuring the energy efficiency of different cache mechanisms. The reason behind this is that the terms are not comparable to each other: the energy term includes the energy spent by the cache only, whereas the delay (or delay^2) term includes processor performance too.

If a mechanism can both decrease energy consumption and increase performance, it is immediately a better solution (or worse if increases energy and decreases performance). The purpose of using an efficiency metric is to measure the efficiency of energy usage in the case of trade-offs. Ideally, not using the hardware in question should not be an option. For example, the energy-delay product is a popular metric for evaluating transistor efficiency. In that case, spending no energy at all causes the transistor to not work, making the delay term infinite, and the energy-delay product becomes undefined. However, when measuring caches, turning off the cache does not make delay infinite as the processor can continue working even if there were no cache. Therefore, turning off the cache always becomes the most efficient energy-saving technique, as the energy-delay product always equals zero (instead of an undefined value).

Another option to make the energy-delay product work with cache efficiency would be to include the energy spent by the processor to the total energy spent by the system. That would prevent the energy term from becoming zero when the cache is turned off and solve the previous problem, but this also introduces the energy efficiency of the cache into the measurement instead of comparing the efficiency of different cache mechanisms only. Consider this case, two different cache mechanisms provide the same processor throughput (therefore having equal delays), but one of them can save 90 percent energy whereas the other one can save 10 percent energy. If the energy spent by the processor is too dominant, the energy-delay product of these two example mechanisms would become very similar although there is a very significant efficiency difference between them.

In order to evaluate the efficiency of cache mechanisms while leaving external factors outside and turning the whole cache off out of the question, we use a modified version of the energy-delay product shown in Equation 3.4 [16]. In this formula, the E term represents the energy spent by the cache only. The second term represents the performance lost due to the cache performance. The third term is essentially a constant term which normalizes the second term.

Table 3.1. Simulated system specifications

Number of instructions simulated	100M
Processor frequency	2GHz
Processor width	4
Instruction scheduling	Out-of-order
Reorder buffer size	128 Entries
Register file size	128 INT, 128 FP
L1 i- and d-cache	16KB, 4-way, 4 cycle latency
L2 cache	256, 128, 64 and 32KB, 8-way, 64-byte line size, 15 cycle latency
Maim memory	64-bit bus width, FRFCFS scheduling policy, 2KB row buffer, 12.8 GB/s bandwidth, 11-cycle column, 25-cycle activate, 10-cycle precharge

$$Efficiency = \quad \cdot \frac{1}{IPC - IPC_{no_cache}} \cdot (IPC_{baseline} - IPC_{no_cache}) \tag{3.4}$$

The following subsections compare turning off portions of cache completely with other static energy-saving techniques in the literature and motivate using an explicit set-based partitioning technique as an energy reduction mechanism. Table 3.1. shows the specifications of the simulated system used to obtain empirical results.

Table 3.2. Power/energy requirements of the simulated L2 caches

Configuration	Static energy	Read energy	Write energy
512-8	326.363 mW	1.36356 nJ	1.46549 nJ
256-8	169.223 mW	1.33042 nJ	1.38199 nJ
128-8	90.5569 mW	1.311 nJ	1.34737 nJ
64-8	52.2441 mW	1.29917 nJ	1.31484 nJ

For the motivation study, which compared Set Partitioning to other energy-saving techniques, four different cache sizes for the L2 cache is used: 256KB (512 sets), 128KB (256 sets), 64KB (128 sets) and 32 KB (64 sets). These configurations are represented by the number of sets they include. Baseline configuration is selected as the largest one among them (512), and all other configurations are compared against the baseline to evaluate their performance in terms of the metrics discussed above.

Table 3.2. shows the energy specifications of the simulated configurations, which were obtained using CACTI 7.0 [17] with a 32nm transistor technology is assumed. Although dynamic energy costs (read/write energy) decrease as cache size decreases, this is mainly due to shortening routes between components and reducing selection complexity. Since dynamically turning off portions of the cache will not allow such drops, all configurations will be evaluated using dynamic energy costs of the baseline configuration.

3.1.1. Cache Decay

Cache Decay is an energy-saving technique that turns off unused cache lines to reduce leakage [7]. In general, cache lines that do not generate a cache hit for a certain time are considered unused and powered off.

In our comparison, Cache Decay is implemented in set granularity and no adaptive approaches were used for simplicity. To reduce complexity and extra power dissipation, we follow the methodology by Kaxiras et al. and implement a global counter which controls the decay mechanism [7]. At every tick of the global counter, per-set decay counters are decremented by one and sets whose decay counters reach zero are considered decayed and powered down. Whenever a cache access (whether it results in a hit or miss) or insertion occurs, the decay counter of the set is reset to a certain value.

After a cache set is decayed and powered down, any access to this set results in a cache miss. Furthermore, cache lines will require a certain amount of time for stabilizing after re-opening. We also evaluated delaying insertion of new lines until decayed lines are stable to be written into, but simulation results showed that this approach causes huge performance losses (up to 50 percent IPC loss) due to stalling. Therefore, a bypass mechanism is assumed for Cache Decay when a new cache line is supposed to be inserted into a decayed set.

Table 3.3. shows the parameters used for evaluating Cache Decay. Decay period represents the duration required for a cache set to be considered inactive and decayed, and the activation period represents the duration required before a decayed set becomes stabilized. As a global counter is used, the activation and decay periods refer to an interval rather than an exact amount of time. For example, a global counter period of 64 cycles and an activation period

of 4 global cycles means that the stabilization of the cache line will occur after 4 ticks of the global counter, which can happen anytime between 192 cycles and 256 cycles.

Table 3.3. Cache Decay parameters

Global counter period	64 cycles
Activation period	4 global cycles
Decay period	4, 128 and 512 global cycles

For each cache size included in Chapter 3.1., four following configurations were simulated: no decay and decay periods of 4, 128 and 512 global cycles (256, 8K, and 32K cycles). Configurations are represented as "number of sets-decay period". For example, 512 represents an L2 cache with 512 sets without Cache Decay, whereas 256-128 represents an L2 cache with 256 sets and a decay period of 128 global cycles.

Applying energy-saving techniques that cause a drop in performance also causes an increase in energy cost as a side effect. This happens due to two reasons: first, decreasing performance causes the execution to take longer, increasing the static energy required to complete a program block. Second, increasing the number of misses causes an increase in the number of cache lines inserted into the cache, therefore causes an increase in dynamic energy. Indeed, the static power requirement of 64-set configuration is 16 percent of the baseline (512-set) configuration. However, simulation results show that the average amount of static energy spent by the 64-set configuration is 20 percent of the baseline, and goes up to 43 percent.

On dynamic energy, shrinking the cache to 64 sets increases dissipation by 27 percent on average, and up to 79 percent. Similarly, applying Cache Decay increases dynamic energy dissipation by 68 percent on average, and up to 141 percent. However, the dominance of static energy dissipation in total energy budget can hide the increase in dynamic energy: the percentage of static energy in total energy consumed varies between 73 percent and 99 percent, with an average of 90 percent.

The average energy spent by all configurations simulated is presented in Figure 3.1. It can be seen from the results that in terms of overall energy reduction, Cache Decay greatly outperforms shrinking cache size: best performing configuration reduces energy

consumption by 70 percent on average (64-set configuration), whereas applying Cache Decay reduces energy consumption by 84 percent. This is to be expected since the reduction achievable by shrinking the cache is limited by the number of sets turned off, while Cache Decay can potentially turn the whole cache off. It is worth noting that Set Partitioning can turn off a variable number of cache sets and overcome this limitation.

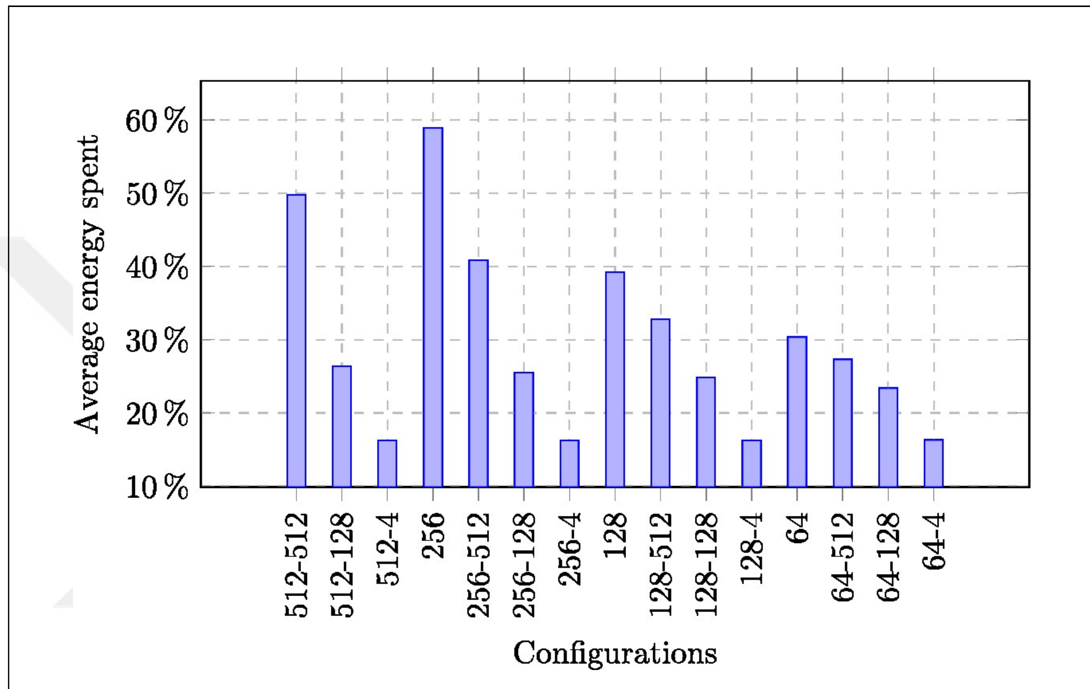


Figure 3.1. The average rate of energy spent by various configurations.

An interesting observation is that applying both techniques can also achieve an energy reduction rate almost equal to the maximum reduction available by these techniques. Applying Cache Decay in 512-, 256- and 128-set configurations reduce the total energy spent by 84 percent. In this sense, it can be said that turning off cache sets does not necessarily impede the energy-saving potential if applied along with Cache Decay.

Applying Cache Decay causes further throughput loss compared to turning off cache sets, which can be seen in Figure 3.2. For all configurations, no-decay achieves the lowest amount of throughput loss in 24 out of 25 benchmarks compared to the baseline configuration. Increasing decay period reduces IPC loss, as it increases the probability of cache lines being reused before decaying, but also reduces energy savings since cache lines will stay on for longer periods once they are accessed and their decay counter is reset. One exception to this

is the case where prolonging the decay period reduces energy savings but does not reduce IPC loss. This exception occurs when cache lines have a reuse distance longer than the decay period and are not accessed before they decay. In this case, keeping cache lines on for a longer time does not improve the cache hit ratio, but increases energy spent to keep those lines on.

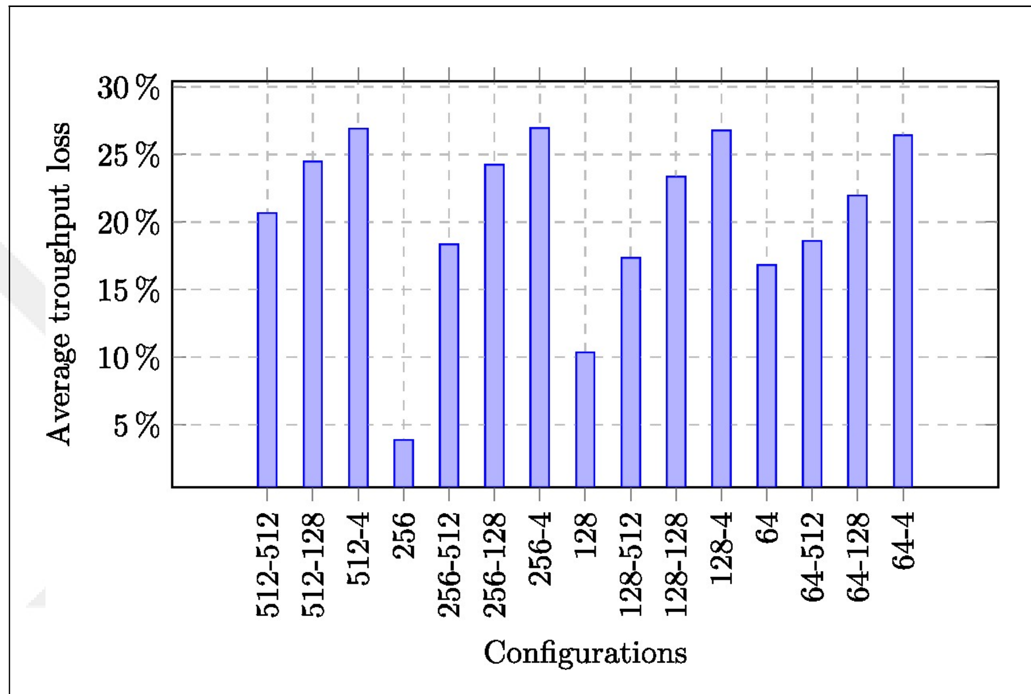


Figure 3.2. Average throughput loss of various configurations.

The trade-off between turning off cache sets and applying Cache Decay is higher throughput of the former and higher energy saving of the latter. Throughput advantage of turning off cache sets comes from its ability to keep data indefinitely, allowing the workload to benefit from these data in a distant future whereas Cache Decay prevents any benefit beyond a certain time. In certain access patterns, especially when the working set of the workload can fit into a portion of the cache, turning off cache sets can outperform Cache Decay beyond dispute by achieving much better throughput while achieving the same level of energy reduction. For example, when running the benchmark bwaves, 512-set configuration with a Cache Decay of 512 global cycles decreases throughput by 9.2 percent while reducing energy by 71.2 percent compared to the baseline. On the other hand, 64-set configuration with no decay decreases throughput by only 0.7 percent while reducing energy by 72.1 percent.

All in all, in terms of energy reduction and throughput loss, applying Cache Decay or not is a question of priority: if reducing energy is more important than achieving high throughput, Cache Decay serves this purpose much better than turning off cache sets. If one's purpose is to save as much energy as possible while keeping throughput high, turning off cache sets is a much more viable option compared to Cache Decay as simulation results show that for 7 benchmarks out of 25, turning off cache sets without applying Cache Decay does save energy while keeping the throughput intact, and 3 benchmarks decrease throughput less than 1 percent compared to baseline. However, applying both techniques constitutes an important option: turning off cache sets while keeping decay period fixed either improves energy savings or keeps it at a similar level, while either decreasing throughput loss or keeping it at a similar level.

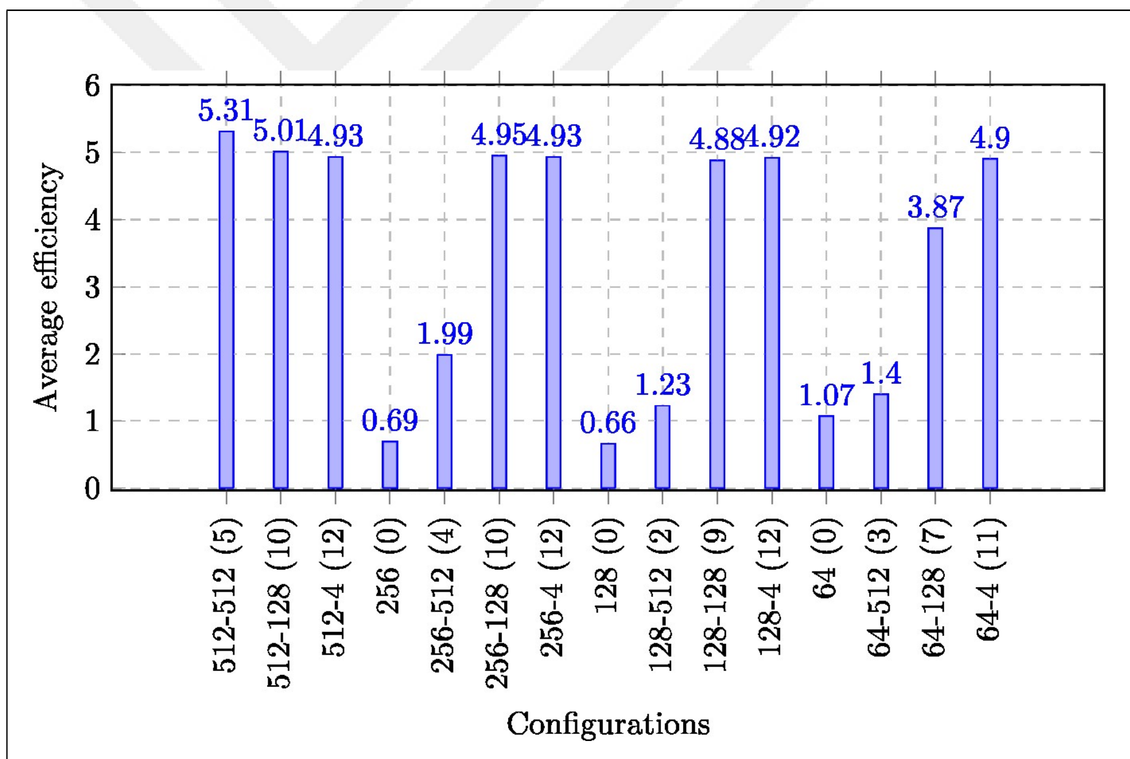


Figure 3.3. The average efficiency of various configurations.

Figure 3.3. shows the normalized average energy efficiency of the simulated configurations. An efficiency value of 1 represents that the given configuration is equally energy-efficient as the baseline. Any value greater than 1 indicates worse efficiency, and any value lower than 1 indicates better efficiency compared to the baseline. Some configurations provide a throughput equal to the no-cache configuration for some workloads, producing an efficiency

value of infinity. To be able to obtain average efficiency values of configurations while being fair, these cases are replaced with an efficiency equal to the worst efficiency produced among the non-infinite configurations (which works for Cache Decay and against turning off cache sets). The number of workloads each configuration produces an infinite energy efficiency can be seen inside the parentheses in the x-axis labels of Figure 3.3.

The efficiency results show that the only configurations that can provide better efficiency than the baseline are the 256- and 128-set configurations without Cache Decay. 64-set configuration has a worse efficiency due to the increased number of conflict misses. Decay configurations produce much worse efficiency due to high throughput losses, close to the no-cache throughput. A decay period of 512 global cycles works more decently compared to the other decay configurations due to its ability to retain data for longer periods of time. Still, all decay configurations perform worse than the baseline configuration in terms of efficiency.

An interesting observation is that Cache Decay with a decay period of 4 always yields a similar level of efficiency. The reason is that the average duration between two accesses to cache lines is greater than the decay period of 4 global cycles for the simulated workloads. This causes lines to decay before they are accessed, converging the hit rate of the cache to 0. This result is the equivalent of having no cache in terms of throughput. Even in that case, a decay period of 4 seems to work more efficiently compared to the other decay configurations, but that is due to the infinite efficiency values being replaced.

One observation is that given any fixed decay period, shrinking the cache size decreases total energy spent by the cache while also decreasing throughput loss. This result is due to the cache access pattern of the simulated benchmarks and the nature of set-based Cache Decay. With a larger cache, accesses spread over a larger area whereas with a smaller cache accesses tend to concentrate on a smaller area. In the absence of a high rate of conflict misses, concentrating accesses to a smaller area works better with set-based Cache Decay. This allows sets to be kept alive for longer and allows benchmarks to benefit from the data retained. This also explains the increase in efficiency as the cache size decreases except 64-128, where conflict misses and data loss works hand in hand to destroy efficiency. However, all configurations still fall short of the efficiency of the 128-set no-decay configurations.

128-set configuration also provides the highest geometric and harmonic mean in terms of 1/efficiency[2].

To summarize, both turning off cache sets and Cache Decay have their own set of advantages over one each other. One advantage of Cache Decay is that it is a more flexible approach: if necessary conditions are met, Cache Decay can adaptively turn off cache sets without affecting the hit rate. On the other hand, turning off cache sets can store data indefinitely and is invulnerable to very long reuse distances. Another difference between the two techniques is that turning off cache sets is more predictable in terms of throughput loss and energy savings, whereas Cache Decay is more susceptible to the access pattern of the instruction stream. Empirically, simulation results imply that Cache Decay is a better option for achieving a greater reduction in energy savings, and turning off cache sets is a better option for maintaining throughput and achieving a higher level of efficiency.

An important intrinsic difference between Set Partitioning and Cache Decay is that although Cache Decay is an energy-saving technique, it is not a cache partitioning mechanism and cannot replace Set Partitioning altogether. In the end, the two techniques are not rival mechanisms, but rather orthogonal techniques which can be applied together.

3.1.2. Drowsy Cache

Drowsy Cache is an energy-saving technique similar to Cache Decay [8]. The difference between the two is that Drowsy Cache can keep the data indefinitely by putting cache lines in a low-power (drowsy) mode rather than turning them off. Although consuming much less energy than traditional cache lines, drowsy lines do dissipate energy unlike decayed cache lines, which are completely powered off. Moreover, drowsy lines must be put back into the awake state before they can be accessed again, which induces an additional latency.

We follow the methodology from Flautner et al. and use the simple policy for evaluating Drowsy Cache [8]. In simple policy, all cache lines are put into drowsy mode periodically. We again follow the original research on Drowsy Cache and use 4000 cycles as the period duration. Whenever a cache hit or an insertion occurs to a drowsy mode, it is restored to the

---

[2] Geometric and harmonic means are vulnerable to few but low instances. Since efficiency is a lower-is-good metric, using these means directly on efficiency yields misleading results. For the geometric and harmonic means, we invert the efficiency metric, turning it into higher-is-better, and then apply the means.

awake mode and stays awake until it is put back into drowsy mode at the beginning of the next period.

Whenever a cache access occurs, all drowsy lines in the set must be awakened to determine whether the requested address is present or not. Similarly, whenever an insertion occurs, all drowsy lines in the set must be awakened to find out which line is going to be evicted. Another approach that alleviates this problem is to decouple the drowsiness of data and tags. This approach allows only tags to be restored in case of a cache access/insert and data to be left untouched, avoiding unnecessary data awakening. The downside of this approach is that it induces another level of latency: upon a cache access, all tags must be awakened (first latency), and then the target line must be awakened according to the tag/recency data comparisons (second latency).

As for timing and energy consumption values, since our simulated configuration and the one in Flautner et al.'s work are different, we use the ratios they provide in their research: the additional latency incurred by the transition from drowsy mode to awake is taken as half the time required for a cache access (both for tag and data). Similarly, the amount of energy dissipated by a drowsy line is roughly 16 percent of an awake line.

Figure 3.4. shows the average energy spent by cache configurations with and without Drowsy Cache. All results presented in this section are in comparison with the baseline configuration. Similar to Cache Decay, Drowsy Cache can reduce energy consumption far more than turning off cache sets (to an extent) can. However, the energy reduction achieved by Drowsy Cache is less than what is achievable by Cache Decay, mainly due to Drowsy Cache, spending some energy to retain data in lines. It can also be seen that the tag-decoupled policy does not provide a significant reduction in energy compared to the Drowsy Cache. This partly due to the increased execution time due to throughput loss, and partly due to the access pattern of the benchmarks. If a benchmark has a low rate of cache access or has a poor distribution over sets, it means that most cache lines will not be awakened anyway.

Drowsy Cache causes a higher throughput loss compared to turning off cache sets on the average, as shown in Figure 3.5. The benchmarks can be divided into two categories for this matter: for the first set of benchmarks, Drowsy Cache can provide a lower throughput loss due to its ability to retain more data. For example, for astar, 512-set Drowsy Cache causes a 2 percent throughput loss whereas halving the cache size causes the 256-set no-drowsy

configuration to cause an 11 percent throughput loss. On the other hand, for the second set of benchmarks where the working set can mostly fit into the cache, Drowsy Cache causes greater throughput loss than turning off cache sets due to the additional access latency it incurs. For example, 128-set configuration provides the same throughput as the baseline for libquantum, whereas 64-set configuration causes a mere 2 percent throughput loss. Applying Drowsy Cache however causes libquantum to achieve 26 percent lower throughput on the best-performing configuration, compared to the baseline.

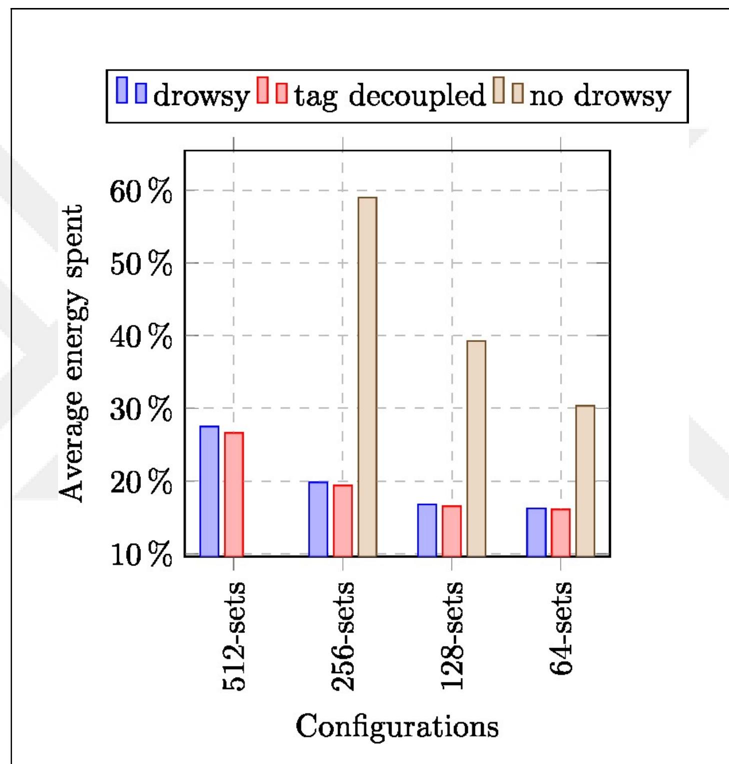


Figure 3.4. The average rate of energy spent by various configurations.

Although omitted, geometric and harmonic means of throughput loss also yield similar results, proving that the higher throughput loss caused by Drowsy Cache is not due to few but very malevolent instances. Even when benchmarks with zero throughput loss are left out (which works against turning sets off), Drowsy Cache causes a greater throughput loss, using either arithmetic, geometric, or harmonic mean.

Results obtained from the simulations show that Drowsy Cache provides less energy reduction, but also causes less throughput loss. This supports the claim that unless there is a leftover area from the working set which can be safely turned off without affecting

performance, there exists a trade-off between energy reduction and throughput. Unless one aims energy reduction or maximum throughput only, the difference emerges in the efficiency of energy reduction.

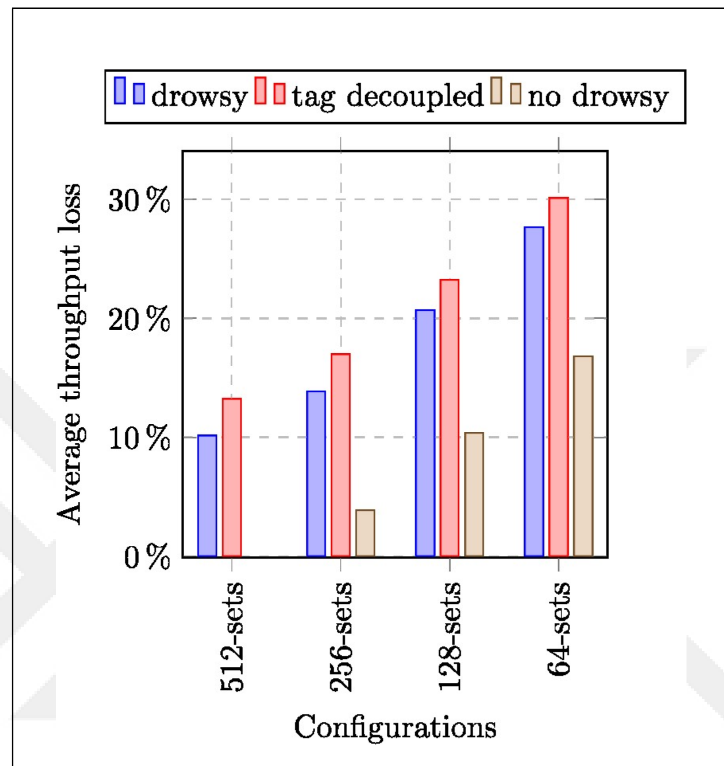


Figure 3.5. The average throughput loss of various configurations.

Figure 3.6. presents the energy efficiency of simulated configurations. The configurations are named using the following convention: configurations which have a “dr” in their name represent the configurations that utilize Drowsy Cache. Configurations which have a “tag” in their name represent the configurations which decouple the drowsiness of tag and data. The numbers inside the parentheses represent how many workloads produce an undefined efficiency value due to division by zero.

In some cases, Drowsy Cache may cause a throughput lower than the no-cache configuration due to higher latency. To obtain valid efficiency values, these configurations are assumed to have a throughput equal to the no-cache configuration. In these cases, the efficiency becomes infinity. To be able to obtain average efficiency values of configurations while being fair, these cases are replaced with an efficiency equal to the worse efficiency produced among the non-infinite configurations (which works for Drowsy Cache and against turning off

cache sets). The number of workloads each configuration produces an infinite energy efficiency can be seen inside the parentheses in the x-axis labels of Figure 3.6.

256- and 128-set non-drowsy configurations provide the best efficiency values. 512-set Drowsy Cache can also provide better efficiency than the baseline, due to its high energy saving and mediocre throughput loss. 256-set Drowsy Cache also provides worse but similar efficiency compared to the baseline. As cache size shrinks even further, the increase in conflict misses combined by the increased latency causes these configurations to degrade in terms of efficiency. This can also be supported by the efficiency difference between drowsy and tag-decoupled configurations: although tag-decoupled configurations provide slightly higher energy savings, the increase in access latency causes greater throughput losses, and eventually worse efficiency levels. 128-set also provides the best geometric and harmonic means in terms of inverse efficiency (see the efficiency discussion in Chapter 3.1.1. for more detail).

Drowsy Cache yields similar advantages/disadvantages compared to Cache Decay compared to turning off cache sets and is somewhat placed in between those two. The Drowsy Cache provides higher energy savings compared to turning off cache, but less compared to Cache Decay. It also provides lower throughput loss compared to Cache Decay and higher compared to turning off cache sets, albeit Drowsy Cache can provide a throughput even worse than having no cache. Larger configurations of Drowsy Cache provide much better efficiency compared to Cache Decay. The presented results may be misleading for smaller configurations where Drowsy Cache seems more efficient, but that is due to infinite values being replaced. In fact, Cache Decay is more efficient than Drowsy Cache in 64-set configurations where it spends less energy and provides greater or equal throughput.

Drowsy Cache and turning cache sets off are not rival, but orthogonal techniques. A smart cache mechanism can utilize both techniques hand-to-hand in order to maximize energy savings and/or efficiency. At worst, such a smart policy can choose to disable one technique while the other is active if it deems utilizing both techniques simultaneously is harmful. In the end, the Drowsy Cache is not a cache partitioning technique and cannot replace Set Partitioning altogether.

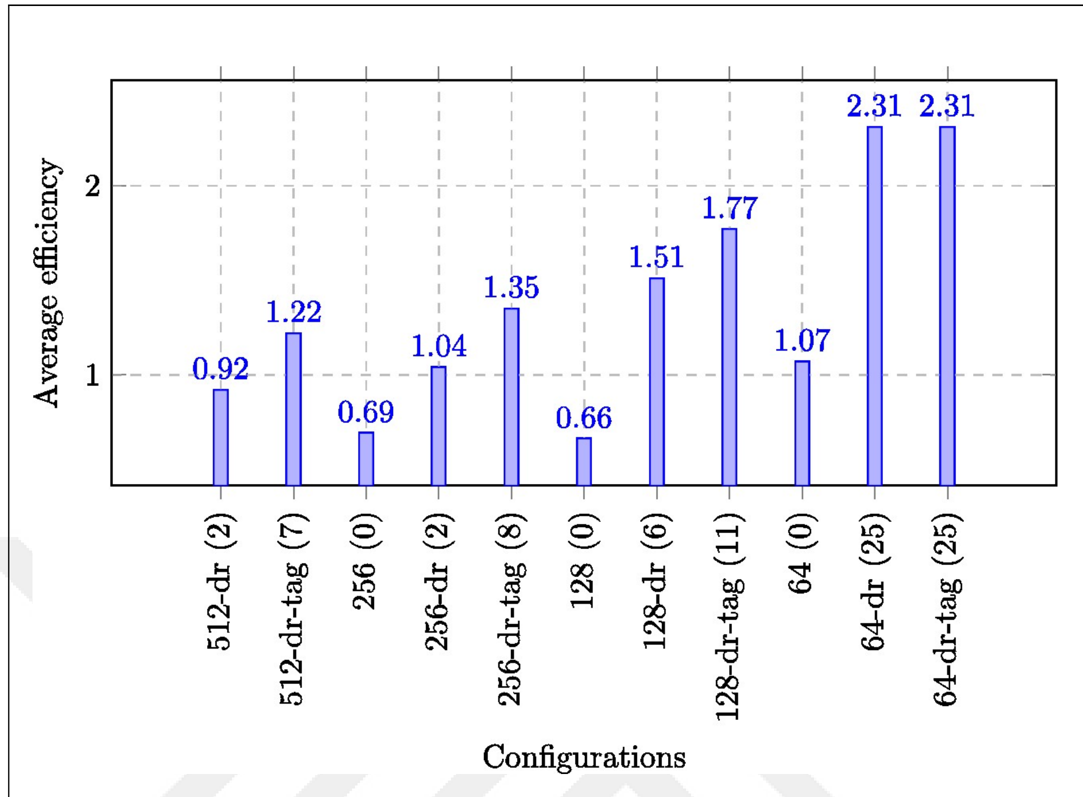


Figure 3.6. The average efficiency of various configurations.

3.2. ADDITIONAL ENERGY COSTS OF SET PARTITIONING

This section examines the additional energy consumption caused by the implementation of Set Partitioning to a cache. Section 3.2.1 examines the characteristics of the Fast Set Redirection hardware in terms of energy consumption and latency. Section 3.2.2 discusses the additional energy cost caused by the Double Access mechanism and manual invalidations.

3.2.1. Fast Set Redirection Hardware

The Set Partitioning mechanism requires the mapping function of cache addresses to cache sets to be altered in a way that supports set-based partitioning. In previous sections, a small circuit named Fast Set Redirection (FSR) was proposed for carrying out the necessary computation to re-map incoming memory addresses. The FSR computation consists of two

parts: shrinking the set index if it is greater than partition size and adding the set index to the beginning set. The algorithm for FSR is shown in Algorithm 3.1.

The FSR computation must be carried out for each cache access, as the memory address must be re-mapped to a new set according to the beginning set and the size of the partition. This causes additional energy consumption for Set Partitioning. The amount of energy spent by the FSR hardware directly affects the energy consumption of Set Partitioning, and thus, its ability to save energy.

In order to derive the energy spent by FSR, the circuit is implemented in Verilog, including the hardware required to synchronize it with the clock signal. Then, the design is synthesized using the Synopsys Design Compiler RTL Synthesis Tool[3].

Algorithm 3.1. Fast Set Redirection

set index = (address) mod (rounded up partition size)

if set index ≥ partition size **then**

    set index = set index – partition size

end if

set index = set index + beginning set of partition

The result of the synthesis shows that the FSR computation can be completed in 0.5ns time (equal to a one-cycle delay) and requires 0.2pJ to complete. According to the energy figures obtained from CACTI (shown in Table 3.2., this amounts to a less than 0.02 percent increase in dynamic energy per access. Similarly, the FSR hardware dissipates around 48uW static power, which is less 0.1 percent compared to the smallest cache configuration evaluated (64-8 configuration in Table 3.2.).

Abousamra et al. propose a similar set-based cache partitioning mechanism, which uses per-core lookup tables in order to redirect incoming addresses to set-based partitions [18]. The structure of such lookup tables is configured in CACTI and static and dynamic energy

[3] The sc12mc_cln40g_base_rvt_c40_ss_typical_max_0p81v_125c library was used for synthesis. This library uses the TSMC 40nm CLN40G transistor model with a nominal voltage of 0.81V. Although the transistor model used by the synthesis tool is slightly different from the one used by CACTI, the energy cost of the FSR circuitry is so small compared to the cache access energy, these differences can be ignored.

consumption caused by these lookup tables are estimated. According to these estimations, a single lookup table causes a 4.4mW static energy, while spending 15pJ for reading an entry. The proposed FSR circuitry spends 75 times less dynamic energy to compute the new set index while dissipating 91 times less power compared to a single lookup table. For the simulated environment, 16 lookup tables will be required, as each core will need a separate table. For this configuration, FSR requires less than 0.1 percent power compared to the lookup table approach. This amount also approaches to the leakage power of a 44KB cache, which is greater than the smallest cache configuration evaluated, and still a considerable amount for large caches.

Both approaches cause an additional one cycle delay, so neither mechanism outperforms one another in terms of additional access delay. The lookup tables will require updates after each allocation, which will cause an extra latency for allocation. However, since the duration of these updates is negligible compared to the partitioning period and can be overlapped with the invalidation process, this negative aspect of lookup tables is ignored.

3.2.2. Other Energy Costs

One element which contributes significantly to the energy consumption is the Double Access (DA) mechanism. At the end of each repartitioning period, the mapping function of any given partition can potentially change, effectively rendering the old data inaccessible. In order to mitigate the performance problems, DA lets the partition try to access the data using the previous mapping function if the access with current mapping function misses. This inevitably causes an increase in the number of cache accesses, also increasing the dynamic energy consumption.

Dynamic energy consumption constitutes only a small fraction of total energy consumption in large caches (as discussed in Chapter 3.1.1., dynamic energy uses only between 1 percent and 27 percent of total energy for the benchmarks tested with the configuration discussed in Chapter 3.1.1.). The DA mechanism doubles a fraction of total cache accesses, therefore doubling a fraction of total dynamic energy consumption. The experimental results show that the DA mechanism causes a 3.88 percent increase in total energy consumption for the simulations discussed in Chapter 4.1.2.4.

Another issue regarding energy is the writebacks caused by bulk invalidations at the end of each repartitioning period. However, due to the length of the repartitioning periods and the low percentage of forced invalidations, these writebacks cause an insignificant increase in total energy consumption, which amounts to less than 0.01 percent of the total energy budget.

All things considered, Set Partitioning causes less than 4 percent increase in total energy spent by the last level cache simulated, while providing a throughput gain of 5.6 percent and a fairness gain of 4.8 percent over Vantage. It is worth noting that the increase in energy is compared to the baseline configuration, where no partitioning is applied, whereas the throughput and fairness gains are compared to Vantage.

Most of the energy cost increase of Set Partitioning comes from the DA mechanism. It can be predicted that as time progresses, more leftover sets from previous periods may get evicted, and the benefits of DA in terms of throughput may diminish. Considering this idea, one may contemplate a modified DA mechanism in which secondary accesses are abandoned after a threshold amount of cycles. However, such an approach would be infeasible since DA also serves a secondary purpose: by utilizing DA, Set Partitioning can access lines which are inserted in the previous period, i.e. it helps to prevent cache lines from becoming lost. If DA is abandoned mid-period, some lines which are retained in the cache may become inaccessible, hurting coherency as discussed in Chapter 2.1.3. Therefore, it is not possible to abandon DA mid-period to mitigate energy losses since it would require Set Partitioning to initiate a new partitioning period along with cache flushes.

3.3. A RULE-BASED APPROACH FOR SET PARTITIONING

Energy-saving techniques for cache structures can be examined under two major classes: dynamic and allocation-based. Dynamic techniques such as Cache Decay and Drowsy Cache can alter the state of cache lines/sets/ways independently upon the occurrence of an event (i.e. cache set not being used for a certain amount of time). On the other hand, allocation-based techniques decide to not use a portion of the cache at allocation time, considering overall system behavior. The proposed allocation mechanism for Set Partitioning is a rule-based one. Hence, the modified Set Partitioning mechanism which also considers energy consumption also becomes a rule-based one (which will be referred to as Energy Classifier from now on).

Set Classifier, the previous allocation policy, aims to optimize overall throughput and does not consider energy consumption. Therefore, it allocates the whole cache space to cores, regardless of energy efficiency. Energy Classifier aims to save energy as much as possible without hurting the throughput and fairness improvements achieved by the Set Partitioning mechanism.

Set Classifier classifies instruction streams into four categories and appoints *weights* to the cores according to their classes. The classifier appoints one weight to harmful streams and two weights to harmless streams (Equation 3.5). Then, the cache space is divided into total weight, calculating how much cache space will be allocated to a single unit of weight (Equation 3.6). Lastly, all cores are allocated cache space according to their weights (Equation 3.7).

$$Total\ weight = 2\ \cdot Number\ of\ Harmless\ threads + Number\ of\ Harmful\ threads \tag{3.5}$$

$$U = \frac{Number\ of\ sets\ in\ cache}{Total\ weight} \tag{3.6}$$

$$Allocation = \begin{cases} floor(2 \cdot U), & if\ thread\ is\ Harmless \\ floor(U), & if\ thread\ is\ Harmful \end{cases} \tag{3.7}$$

The difference between Energy Classifier and Set Classifier is that Energy Classifier applies a lower bound to the total weight, which will be referred to as LIMIT from on. If the total weight is smaller than LIMIT, it is replaced with that value (Equation 3.8).

$$Total\ weight = \begin{cases} LIMIT, & if\ Total\ weight\ \leq LIMIT \\ Unchanged, & otherwise \end{cases} \tag{3.8}$$

If the total weight at allocation decision is already equal to or greater than LIMIT, it is assumed that there is sufficient need for cache space to keep the whole cache powered, and Energy Classifier behaves exactly as the Set Classifier. The difference between the two allocation policies emerges when the total weight is smaller than LIMIT. In that case, it is assumed that there is not sufficient need to use the whole cache, and the cache will be divided into more sub-portions than the initial total weight. Since cache space is allocated to cores based on their weight, some portions will be left unallocated, allowing them to be powered down until the next allocation decision.

An example of Energy Classifier is given as follows: assume the cache is partitioned among two cores, which are classified as harmless and harmful, respectively. The total weight of the cores will then become 3 (2 for the harmless, 1 for the harmful core). If the LIMIT is 4, the cache space is divided into four equal units. Since the harmless core has a weight of two, it gets two units of space, effectively allocation half of the cache space to it. The harmful core gets one unit of space, allocating it one-fourth of the cache. The remaining one-fourth of the cache is left unallocated and is powered down until the next allocation decision in order to save power.

The value of LIMIT determines how aggressive Energy Classifier is in terms of trading throughput over power. If LIMIT is equal to the number of cores, it means that the allocation policy assumes a baseline cache need where all cores are running harmful-class applications and will save power only when the total cache need is lower than that. If the LIMIT is equal to twice the number of cores, it means that the baseline cache need is assumed to be equal to the case where all cores are running harmless-class applications, and so on. In general, a lower LIMIT value will make it more unlikely for the initial total weight to be smaller than that, meaning the allocation policy will be less likely to save power by not allocating cache space.

4. TESTS AND RESULTS

This chapter presents the empirical results obtained by simulation which evaluates Set Partitioning int terms of throughput, fairness and energy consumption. Section 4.1 presents and discusses the results for evaluating Set Partitioning in terms of throughput and fairness. Section 4.2 evaluates a modified version of Set Partitioning as an energy-saving technique in terms of throughput and energy efficiency.

4.1. SET PARTITIONING

This section presents the simulation results obtained for evaluating Set Partitioning. Section 4.1.1 describes the experimental methodology used for obtaining the empirical data. Section 4.1.2 discusses the simulation results that show the impact of Set Classifier, Double Access mechanism, manual invalidations and Fast Set Redirection hardware in terms of throughput and fairness, as well as the overall impact of Set Partitioning in comparison with Vantage partitioner, state-of-the-art in the literature.

4.1.1. Experimental Methodology

Set Partitioning is tested in MacSim simulation environment [19]. Thirty 32-core workloads are created, twenty-five workloads are constructed by randomly selecting four benchmarks and repeating each benchmark eight times, and five workloads are constructed by randomly selecting eight benchmarks and repeating each benchmark four times. Intel Pin tool is used to create executable traces for SPEC2006 benchmarks [20,21]. List of workloads used are given in Table 4.1.[4] All benchmarks which did not produce any errors in MacSim simulation environment are used. The excluded benchmarks are perlbench, gcc and cactusADM. Benchmarks start running from their respective regions of interest. When a benchmark finishes executing its region of interest, it is re-run from the beginning of its trace.

[4] Benchmark abbreviations are: astar(A), bwaves(B), bzip2(BZ), calculix(C), deal(D), games(G), gems(E), gobmk(K), gromacs(R), hmmer(H), href264(F), lbm(L), leslie3d(L3), libquantum(Q), mcf(M), milc(I), namd(N), omnetpp(O), povray(P), sphinx(S), sjeng(J), soplex(SO), tonto(T), wrf(W), xalanc(X), and zeusmp(Z).

Table 4.1. Workloads

1	T-X-O-W	16	B-R-Z-L3
2	T-P-N-X	17	D-R-L-G
3	T-H-K-M	18	J-T-G-W
4	C-X-W-SO	19	A-H-T-D
5	W-S-L-O	20	SO-H-D-P
6	P-I-C-D	21	O-M-D-W
7	N-J-G-S	22	N-M-SO-BZ
8	D-R-O-C	23	C-T-Z-L3
9	H-B-X-D	24	D-R-X-M
10	F-BZ-K-L	25	C-M-B-SO
11	H-E-R-C	26	C-H-M-L3-B-X-SO-K
12	C-E-P-L	27	F-E-BZ-S-A-K-D-L
13	C-M-F-P	28	D-K-T-R-H-O-J-C
14	K-P-G-S	29	Q-B-BZ-R-BZ-Z-W-L
15	T-L3-K-E	30	W-T-Z-P-I-N-SO-X

In the simulated configuration, each processor core has private L1 instruction and data caches, and all cores share a single L2 cache. As L1 caches are private, no partitioning is needed there. The study focuses only on a shared L2 cache. Simulated processor configuration is shown in Table 4.2., and partitioning algorithm parameters are given in Table 4.3.. Simulations are carried out for 100M cycles where the warmup duration is 10M cycles and classifier period length is 5M cycles for evaluating individual mechanisms (Chapter 4.1.2.1., Chapter 4.1.2.2., Chapter 4.1.2.3.). Simulations for obtaining the results presented in Chapter 4.1.2.4. are carried out for 500M cycles where warmup duration and classifier period length are 50M cycles each.

Table 4.2. Processor configurations

Number of cores	32
Processor core	8-wide, 256 entry ROB, 1 thread per core
Private L1 icache and dcache	64 sets, 4 ways, 64-byte line size, 3 cycle access latency
Shared L2 cache size	2048 sets, 16 ways, 64-byte line size, 23 cycle access latency, non-inclusive-non-exclusive
Main memory	4-wide bus, 9 columns, 90 cycle activation, 90 cycle precharge, 16 banks, 8 channels, 2048 byte rowbuffer size, FRFCFS scheduling policy
Simulation duration	100M (500M) cycles
Warmup duration	10M (50M) cycles

Table 4.3. Parameters of the partitioning algorithm

Classifier period length	5M (50M) cycles
Traffic rate lower threshold	0.001
Traffic rate upper threshold	0.006
Miss rate lower threshold	0.5
Miss rate upper threshold	0.75
Decay rate lower threshold	0.5
Decay rate upper threshold	0.75
Decay reset value	4
Decay period length	5000 cycles

Set Partitioning is evaluated based on two metrics: throughput and fairness. Throughput represents cumulative IPC of all cores, whereas fairness represents the harmonic mean of all applications' rate of throughput compared to their standalone performance, as shown in Equation 4.1 [22].

$$Harmonic\ mean = \frac{n}{\sum_{i=1}^{n} \frac{IPC_{standalone}}{IPC_{share}}} \quad (4.1)$$

Lookahead is used as the allocation policy when evaluating Set Partitioning against Vantage partitioning, as well as evaluating the proposed Set Classifier [5]. For Vantage and Set Classifier evaluations, 16-way UMON data is interpolated to 256 data points for as proposed by Sanchez and Kozyrakis [5].

4.1.2. Results and Discussion

The proposed Set Partitioning has three major improvements over naive set-based partitioning: Set Classifier (SC), Double Access (DA) and Fast Set Redirection (FSR) mechanisms. The former two aim to improve the performance of Set Partitioning whereas the latter one is required for making Set Partitioning feasible. In this section, the individual contribution of each improvement is analyzed. Then, an empirical analysis of FSR's ability to distribute accesses, in addition to the simulation results is presented. Finally, to show overall performance gains and to compare Set Partitioning against other work proposed in the literature, Set Partitioning is compared against the Vantage scheme.

4.1.2.1. Set Classifier

Throughput and fairness gains of Set Classifier compared to interpolated Lookahead is shown in Figure 4.1. Set Partitioning performs better in terms of throughput compared to interpolated Lookahead on 18 workloads out of 25, achieving a throughput gain of 8.1 percent on the average, peaking at 33.3 percent. In terms of fairness, average and peak gains reach 5.6 percent and 53.9 percent, respectively.

Set Classifier works better than interpolated Lookahead since it is a tailor-made design targeting Set Partitioning. UMON structures in the Lookahead algorithm exploits the fact that LRU obeys stack property [23]. That allows Lookahead to find out how many cache hits an application will gain when additional cache ways are allocated. However, the same rule does not apply to cache sets, and Lookahead cannot determine cache hit gain per allocated sets as accurately as it does for cache ways. Besides, in addition to cache access and miss information, Set Classifier also leverages decay information, which provides hints on what portion of a cache area is being actively used. In other words, decay information gives the classifier hints on when it is safe to take cache sets away from cores. The number of cache accesses shows how intensively an application uses the cache but does not provide information on how diverse these accesses are. Decay information, on the other hand, is affected by the distribution of accesses among sets. For example, two applications may heavily access the cache, but accesses of one application may be concentrated on a handful of sets whereas accesses of the other may be distributed more evenly. In such a case, while taking sets away from the first application may be considered safe (since it is not using all of its sets anyway), taking sets away from the second application may severely degrade overall system performance. Decay information allows Set Classifier to distinguish application behavior in such cases.

4.1.2.2. Double Access Mechanism

Dynamic partitioning of caches requires partition sizes (or their equivalent terms, such as the insertion position in PIPP [2]) to be periodically changed. With Set Partitioning, this means changing the number of sets dedicated to a partition is quite common. Therefore, whenever partition sizes are changed, the mapping function also changes, and all data

suddenly become inaccessible to all partitions. The only exception is when a threads both beginning set and partition size remains unchanged.

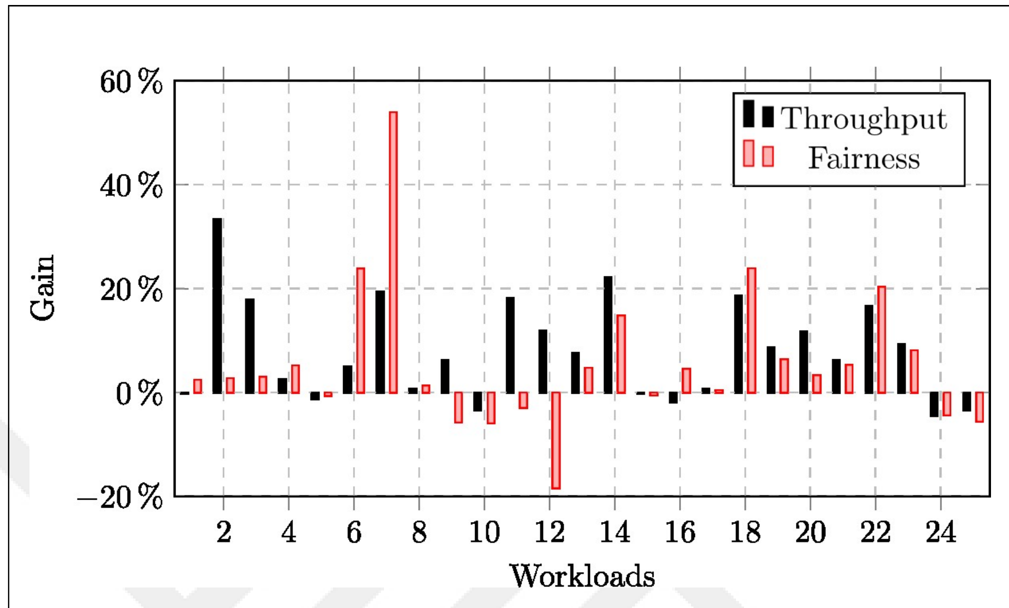


Figure 4.1. Throughput and fairness gain of Set Classifier.

DA mechanism allows partitions to retain access to their old data until they are evicted by the new owner of their respective positions. DA initiates a secondary access if a line cannot be found upon the initial access. Naturally, if a line is found during secondary access, its latency becomes twice the regular latency of the cache.

DA reclaims some of the throughput loss caused by the artificial flushes which take place at the end of each period. Figure 4.2. shows the throughput and fairness gains of Set Partitioning when DA is utilized. 24 out of 25 workloads perform better when DA is utilized, and Set Partitioning performs 2.4 percent better with DA, on the average, peaking around at 9.4 percent. DA also provides an extra 3 percent fairness on average and a peak fairness gain of 18.5 percent.

The effect of extra latency induced by secondary access can be examined by comparing the DA mechanism with an idealistic mechanism called DA-Parallel, which simultaneously initiates both primary and secondary accesses without any penalties. As expected, DA-Parallel performs better than DA, but only 0.4 percent better in throughput and 0.1 percent in fairness on average. However, since DA-Parallel initiates two accesses at the same time, it would require the cache to have twice as many ports to carry out these simultaneous

accesses. Hence, DA-Parallel becomes an infeasible option when its greater complexity and very limited gains are considered.

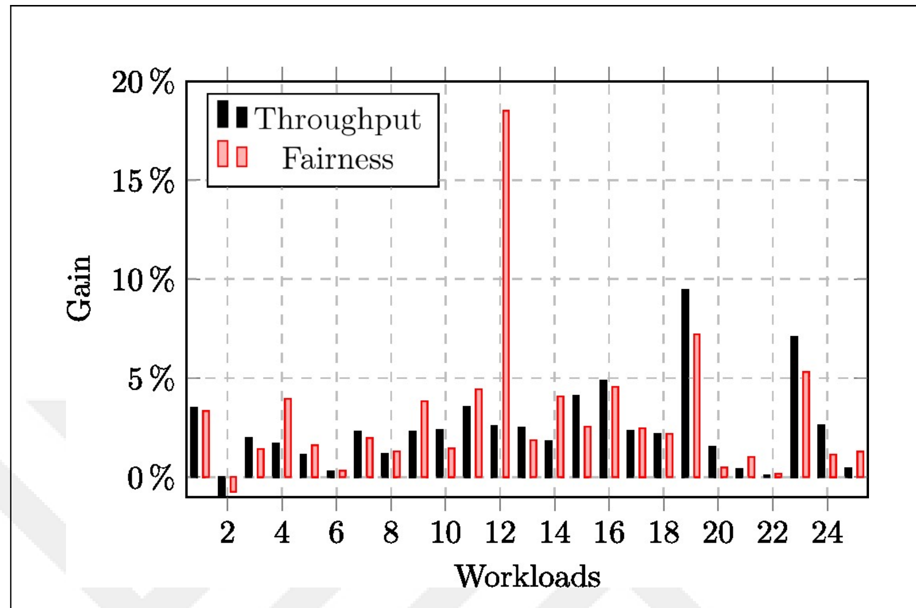


Figure 4.2. Throughput and fairness gain of Set Partitioning with DA over without DA.

In order to examine the effect of cache lines becoming inaccessible due to repartitioning, another idealistic mechanism is also investigated, which copies all data in the cache to a buffer and reinserts all lines into the cache using the new access functions with zero time penalty. This mechanism provides a 2.8 percent throughput gain. The throughput gain obtained by DA, which is 2.4 percent, shows that DA can reclaim most of the throughput loss caused by changing the access function at each repartitioning period. Interestingly, this reinsertion mechanism provides a 2.3 percent gain in fairness compared to the 3 percent gain provided by DA. This is caused by the nature of both mechanisms: reinsertion mechanism may evict some data without the immediate need if the space allocated to a partition shrinks after repartitioning, while DA helps partitions to virtually have greater cache space than they are allocated by allowing them to access data from other partitions until these lines are evicted gradually.

4.1.2.3. Fast Set Redirection

The proposed FSR mechanism computes the set index by rounding partition size up to the closest power of two and applying a bitmask. Resulting values, which are greater than or equal to the partition size, are directed back inside to the partition by subtracting the partition size from them. This causes first *power - size* sets to be accessed twice as much compared to the rest due to accesses to the interval [size, power] being directed to [0, power - size], under the assumption of a uniform distribution of addresses.

Set Partitioning preserves associativity by partitioning the cache by sets. The number of additional conflicts caused by both decreasing number of sets and FSR is usually not high enough to cause a significant throughput loss. Figure 4.3. shows the throughput and fairness gains of Set Partitioning with FSR. Note that both Set Classifier and DA are utilized in these tests. According to these results, FSR causes a throughput loss of 0.7 percent and a fairness loss of 0.4 percent, on the average. 16 out of 25 workloads perform worse with FSR. As expected, throughput loss caused by FSR is small and certainly worth making the sacrifice for the throughput gain obtained by utilizing Set Partitioning. It is worth noting that six workloads perform better when FSR is utilized. FSR can lead to throughput gains by either a) redirecting accesses to different sets where evicted lines will be less critical than where modulo operation would direct to, or b) causing allocation decisions to change because some applications perform worse in previous epochs. Although both cases are possible as seen in Figure 4.3., these phenomena are purely coincidental and are not to be relied upon.

In addition to IPC gains obtained from simulation runs, the results of a statistical analysis of FSR is also presented. In this analysis, 30 different cache sizes (in terms of the number of sets) in the interval [1, 10, 24] were picked, where all sizes are generated randomly except for the 30th, which was hand-picked as 384 as it is right in the middle of two full exponents of two. Under the assumption that addresses of cache accesses are uniformly distributed in the address space, 100K 32-bit addresses were generated randomly for each cache size.

To compare the effectiveness of the modulo operator and FSR, two metrics were used: the entropy of the cache and the number of conflicts that occurred. The entropy of an operator is computed using Equation 4.2 where p_i represents the probability of an address being mapped to ith set and is calculated as in Equation 4.3.

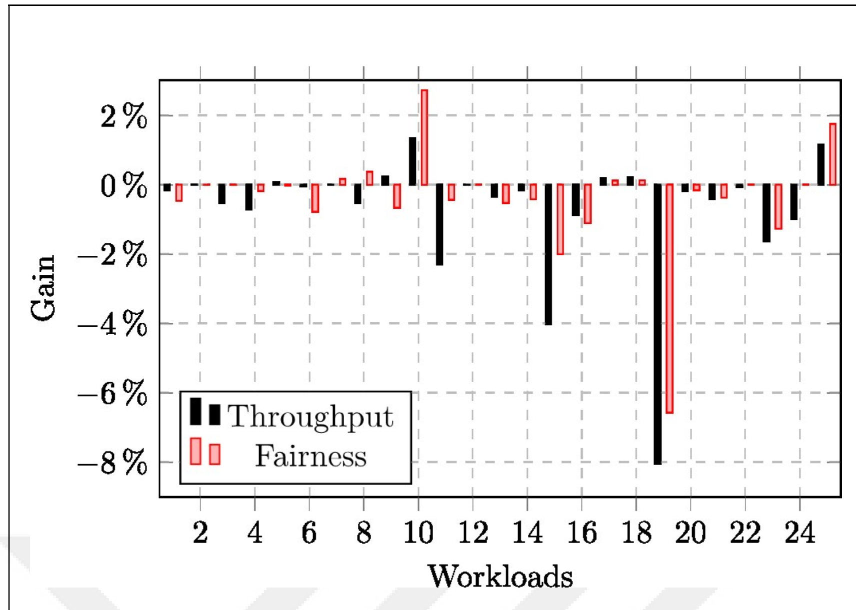


Figure 4.3. Throughput and fairness gain of Set Partitioning with FSR over without FSR.

$$Entropy = \sum_{i=0}^{number\ of\ sets-1} p_i \cdot log_2(p_i) \tag{4.2}$$

$$p_i = \frac{Number\ of\ addresses\ mapped\ to\ i^{th}\ set}{Number\ of\ accesses} \tag{4.3}$$

Higher entropy means addresses are mapped out to available sets in a more uniform fashion, whereas a lower entropy means addresses tend to be mapped in fewer sets with a higher density. The purpose of the mapping function is to reduce conflicts as much as possible by using all sets equally. Therefore, mapping functions seek out to maximize the entropy value. Hence, entropy can be regarded as a good indicator of a mapping function's performance. Figure 4.4. and Figure 4.5. show the entropy gain and the increase in the number of conflicts of FSR compared to the modulo operator for an artifical input set. Results presented in Figure 4.4. and Figure 4.5. are sorted in an ascending fashion.

Results show that the highest entropy loss is 1.4 percent. However, when addresses are generated randomly in a 32-bit address space, it is highly unlikely to have two consecutive accesses to a set will have the same address. Therefore results obtained from this approach resemble the behavior of a direct-mapped cache and do not consider associativity. The fact is that even if unique addresses may be distributed uniformly, uniform distribution of cache accesses can only be assumed for applications which have a streaming-like behavior. Applications that leverage caches have patterns in their cache access footprint.

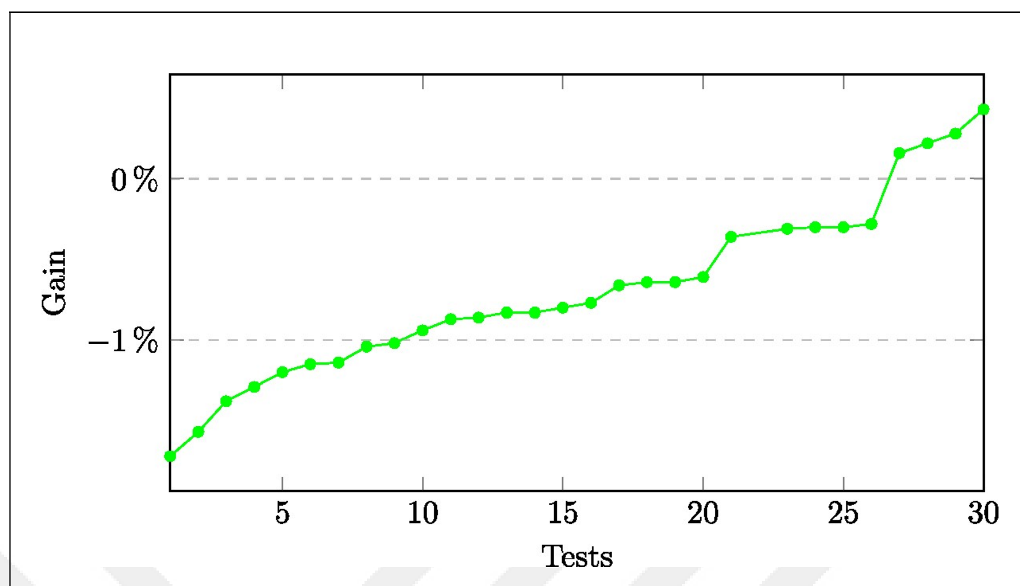


Figure 4.4. Entropy gain for FSR compared to modulo operator.

In the light of these observations, 11 benchmarks were selected and addresses of first 100K cache accesses were used to perform the entropy and cache miss gain tests for caches with associativity of 4 and 8. For each benchmark, the same set of 30 cache sizes were tested. Figure 4.6. and Figure 4.7. show entropy gain and increase in cache misses. Note that entropy is only affected by the number of sets and the mapping function, thus entropy results for associativity values 4 and 8 are the same.

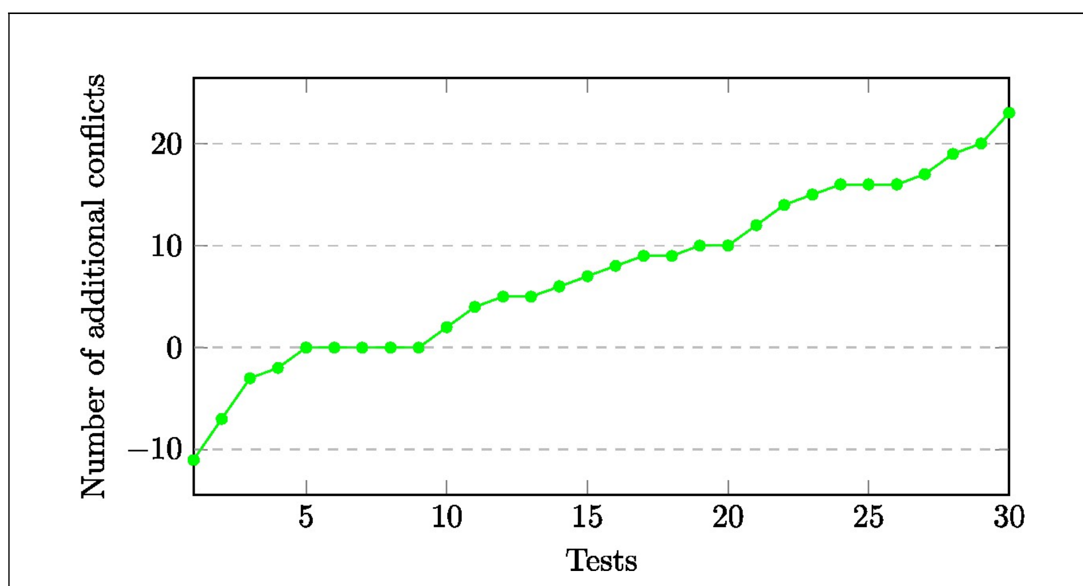


Figure 4.5. Additional conflicts for FSR compared to modulo operator.

On average, FSR has 1.5 percent less entropy than the modulo operator, where the peak loss is 7.3 percent with astar. The increase in the number of cache misses are 9.6 percent and 5.1 percent where associativity is 4 and 8, respectively. Again, peak losses are experienced in astar with 66.2 percent and 49.4 percent. The reason for astar having significantly high entropy loss and an increase in cache misses is that astar has significantly lower entropy in moduli of powers of two after offset bits are eliminated. Especially in cases where the number of sets is a prime number, the modulo operator can leverage the primeness of cache size whereas FSR still uses a power of two for mapping. Indeed, all extreme cases take place where cache size is prime (308 percent increase in cache misses for 19 cache sets, 166 percent for 197 and 548 percent for 97, 708 percent for 37 and 168 percent for 183). When only even cache sizes are considered, the average increase in cache misses becomes 4.8 percent and 0.03 percent for astar, and 4 percent and 0.03 percent for all benchmarks, where associativity is 4 and 8, respectively. It is worth noting that these entropy loss and increase in cache misses are relative to an unrealistic case where the modulo operator for prime numbers is utilized with a latency equal to the bit masking.

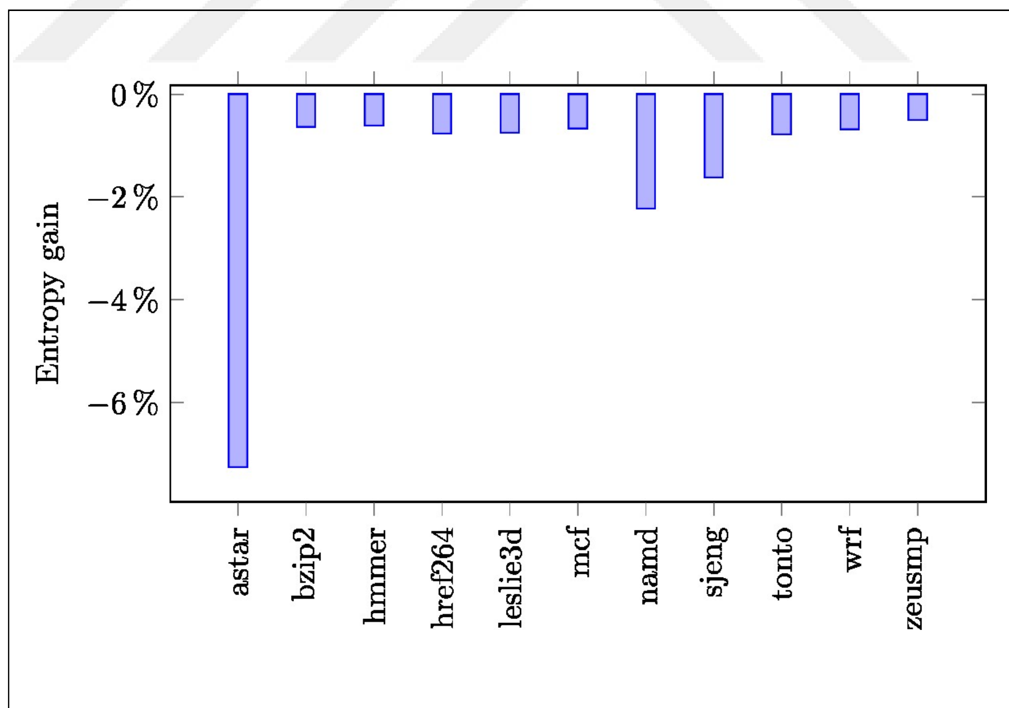


Figure 4.6. Average entropy gain for FSR compared to modulo operator.

4.1.2.4. Set Partitioning

For the workloads tested, Set Partitioning performs better than Vantage in 25 out of 30 workloads in terms of throughput and 23 out of 30 workloads in terms of fairness, as presented in Figure 4.8. The throughput gain of Set Partitioning is 5.6 percent compared to Vantage on the average and reaches up to 33.2 percent, whereas fairness gain is 4.8 percent on the average with the peak gain of 23.3 percent.

Vantage achieves target partitioning by demoting one cache line per miss on the average and adjusting demotion probabilities for applications according to their actual and target sizes. However, this does not guarantee that the number of eviction candidates will be equal to the associativity of cache, as opposed to Set Partitioning. In some cases where no demoted cache lines are found, data from an application whose actual size is smaller than its target size may be evicted. Set Partitioning, on the other hand, guarantees explicit partitioning where data of an application is safe from eviction by other applications.

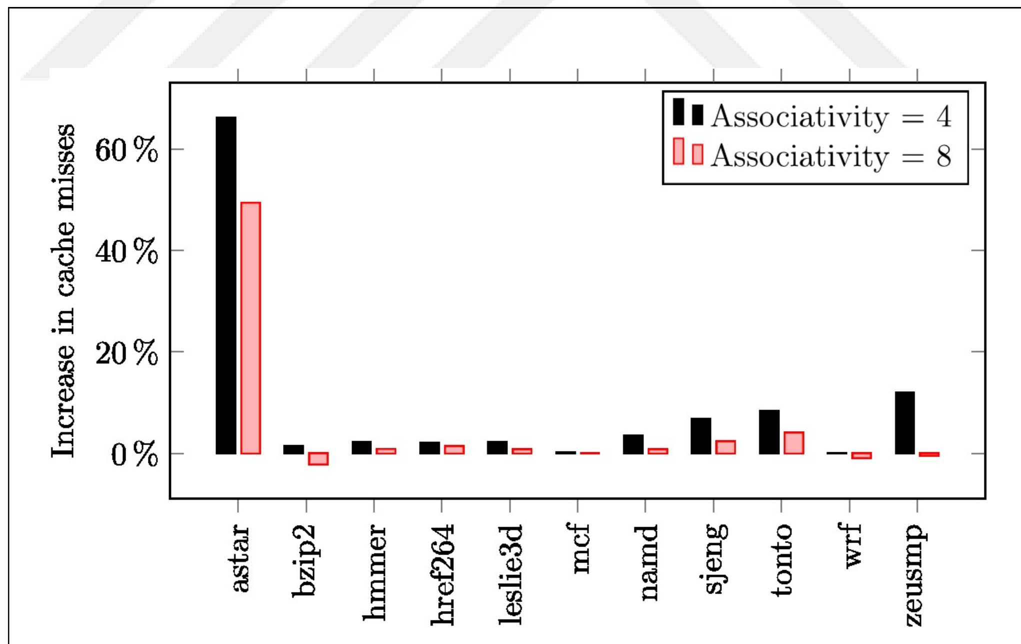


Figure 4.7. Average increase in cache misses for FSR compared to modulo operator.

The purpose of partitioning techniques is to enforce a cache allocation decision upon cores. It goes against this purpose when a core which has less actual size than it's target size loses cache lines. Let *unfair evictions* denote such evictions where the evicted line belongs to a

core that is under its target size. Unfair evictions include evictions caused by both other cores and the core itself. A high number of unfair evictions imply that lines belong to a core are continuously evicted despite allocation decisions try to assign more resources to the same core. Thus, a high number of unfair evictions is a negative indicator of how well a partitioning technique fulfills the allocation decision.

A case study reveals that although there are other factors such as the ability of an allocation policy to make good decisions, cache behavior of the application and the cache usage pattern of concurrently running applications; performance loss of Vantage compared to Set Partitioning is mostly related to unfair evictions. Vantage does not take into consideration whether the evicted line belongs to a core under or over its target size and it simply evicts one of the demoted lines, which is expected to enforce allocation decisions on the average. For the workloads examined, for almost all cores where Vantage performs worse than Set Partitioning, the ratio of unfair evictions to the total number of evictions is much higher than the cores where Vantage performs better. For example, in workload 1, average unfair evictions to total evictions rate has an average of 57 percent for the 14 cores that Set Partitioning improves performance and the average rate for the remaining 18 cores where Vantage performs better is only 1 percent. Unfair eviction rate varies in a large interval between 40 percent and 75 percent in our case studies.

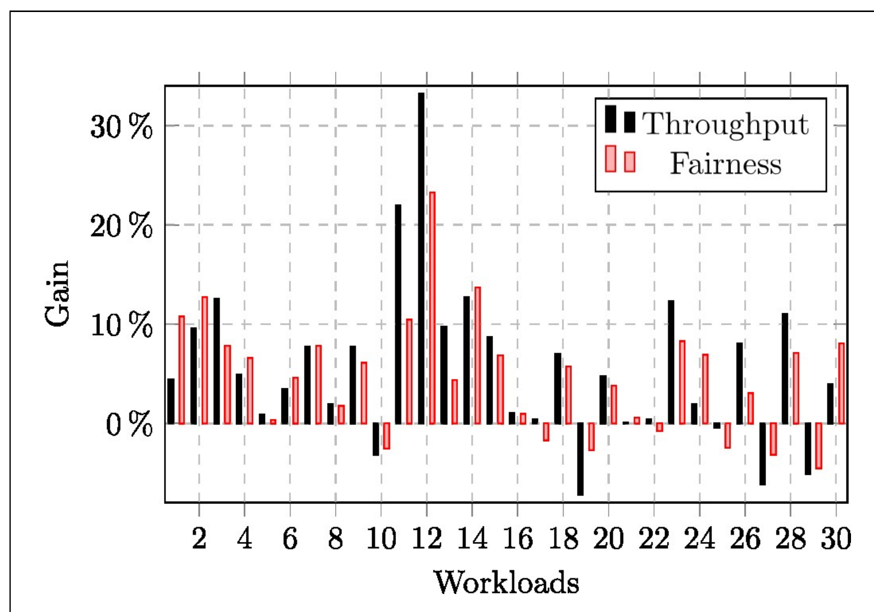


Figure 4.8. Throughput and fairness gains of Set Partitioning over Vantage.

Performance results presented in Figure 4.8. include the cost of invalidations and evictions carried out at the end of each repartitioning period. During this process, every cache line is sequentially checked for invalidation/eviction. One cycle latency for each check is assumed, and the process completes when all dirty stale lines are written back to the main memory.

To mitigate the effects of evictions, the length of repartitioning periods is kept relatively high. In the simulated configuration, in the worst-case scenario (where every line in the cache is written back to the main memory), the time required for invalidation and eviction process takes only 3 percent of the repartitioning period. Set Partitioning remains viable even in such a worst-case scenario: throughput and fairness gain of it over Vantage becomes 3.1 percent and 4.5 percent (compared to 5.6 percent and 4.8 percent), on the average, respectively.

In the studied workloads, the average number of cache lines which are forcefully evicted at the end of repartitioning periods amount to 2 percent of the cache, whereas the maximum value reaches 35 percent. The cost of invalidations are negligible: average throughput and fairness of Set Partitioning would be only 0.05 percent higher if invalidations and evictions caused no latency, and the workload which suffered the highest amount of dirty stale lines lost only 0.2 percent throughput and fairness.

Set Partitioning requires incoming memory addresses to be remapped to the owners partition. To carry out the remapping, Fast Set Redirection (FSR) is introduced. FSR requires an additional 12-bit addition and a layer of parallel 2-to-1 multiplexers. Set Partitioning is evaluated with the assumption that the additional computation required for FSR does not increase cycle time or access latency. However, Set Partitioning is also evaluated for the case where FSR increases access latency from 23 cycles to 25 cycles. The additional two-cycle latency introduced to access latency causes Set Partitioning to perform 0.27 percent and 0.22 percent worse on the average, in terms of throughput and fairness respectively.

4.2. ENERGY CLASSIFIER

This section presents and discusses the impact of Energy Classifier in terms of energy consumption, throughput, energy spent per instruction and energy efficiency. Section 4.2.1 presents the experimental methodology used for obtaining the empirical data presented in this section. Section 4.2.2 presents and discusses the empirical results.

4.2.1. Experimental Methodology

The proposed Energy Classifier is evaluated in terms of throughput, energy consumption and energy efficiency compared to the Set Classifier. The difference between Set Classifier and Energy Classifier yields in the LIMIT. In the experiments, the evaluated LIMIT values are 1x, 2x and 4x the number of cores. The specifications for the simulation environment are given in Table 4.4.

The workloads are generated from all SPEC2006 benchmarks which did not produce errors running in the MacSim simulation environment [19,21]. The excluded benchmarks are perbench, gcc, mcf and cactusADM. The list of workloads is shown in Table 4.5.

Table 4.4. Specifications of the simulation environment

Number of cores	16
Warmup duration	50M cycles
Total duration	500M cycles
Partitioning period	50M cycles
Private L1 i- and d-caches	64KB, 4-way, 4 cycle latency
Shared L2 cache	2MB, 8-way, 26 cycle latency
Main memory	4-wide bus, 9 columns, 90 cycle activation, 90 cycle precharge, 6 banks, 8 channels, 2048 byte row-buffer size, FRFCFS scheduling policy
Processor core	2GHz clock frequency, 4-wide issue, 256 entry ROB, out-of-order execution

Measuring the total energy spent by a configuration for a certain number of cycles does not yield any meaningful results: a configuration can turn the whole cache off and provide the maximum possible energy saving. What is relevant is how much energy a configuration spends in order to finish a task (i.e. in order to complete a certain number of instructions).

However, the simulations are chosen to be run for a certain amount of simulated cycles instead of being run until all benchmarks complete a certain number of instructions for this study. The reason behind this choice is the time limitation. There is a significant variation between the throughputs of the benchmarks, and most benchmarks provide a low throughput when in this study's experimental setup. These factors cause a great increase in simulation

times if one chooses to simulate the system until all benchmarks complete a meaningful amount of instructions in all workloads.

Table 4.5. List of workloads

zeusmp-sjeng-sphinx-hmmer	deal-zeusmp-namd-gamess
lbm-bwaves-bzip2-libquantum	gromacs-omnetpp-zeusmp-milc
namd-bwaves-bzip2-gems	calculix-gromacs-zeusmp-omnetpp
gamess-namd-gems-bzip2	deal-astar-lbm-omnetpp
bwaves-soplex-wrf-libquantum	calculix-xalanc-lbm-leslie3d
gobmk-lbm-leslie3d-wrf	gobmk-deal-gamess-sphinx
gromacs-soplex-gems-wrf	astar-gobmk-sjeng-leslie3d
omnetpp-lbm-namd-sjeng	gobmk-astar-calculix-xalanc
zeusmp-omnetpp-libquantum-hmmer	deal-gromacs-calculix-gobmk
games-leslie3d-namd-sjeng	astar-gromacs-xalanc-zeusmp
astar-calculix-milc-milc	calculix-gobmk-xalanc-gamess
zeusmp-omnetpp-lbm-lbm	astar-deal-gromacs-soplex
xalanc-lbm-sjeng-leslie3d	

Therefore, simulations are run for a certain number of cycles in order to be able to finish those simulations in a feasible amount of time. Then, the amount of energy spent by each configuration is evaluated by finding how much energy is spent per instruction (Equation 4.4). By using this metric, a rough estimation of energy spent by each benchmark until a certain task is completed can be achieved.

$$E/Inst = \frac{Total\ energy\ spent}{Number\ of\ instructions\ completed} \tag{4.4}$$

4.2.2. Results and Discussion

In this section, the results obtained from simulation for Energy Classifier in terms of energy, throughput and efficiency are presented and discussed. The primary goal of Energy Classifier is to save energy by turning off unneeded portions without hurting the throughput. Figure 4.9. shows how much energy is spent by Set Partitioning when Energy Classifier is utilized as the allocation policy, compared to the baseline Set Partitioning where the allocation policy is the Set Classifier. Five different LIMIT values are evaluated. Configurations are represented with the letter L followed by a coefficient, which is

multiplied by the number of cores. For example, L2 represents the configuration where the LIMIT is equal to twice the number of cores.

The expected outcome is that as the LIMIT value increases, the energy consumption should decrease due to Energy Classifier. This is because as LIMIT increases, Energy Classifier divides the cache into more sub-partitions, increasing the chance of leaving leftover sub-partitions not allocated, thus not spending energy. It can be seen from the results that for all workloads simulated, L0.5 achieves a 2.15 percent decrease in energy, while L1, L1.5, L2 and L4 achieve 9.68 percent, 20.89 percent, 32.45 percent and 54.17 percent energy savings, respectively.

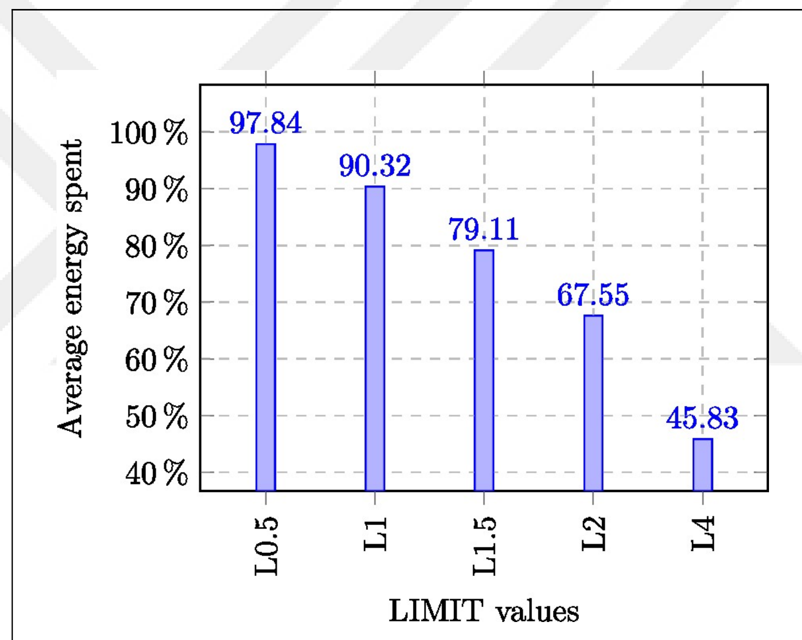


Figure 4.9. The average energy spent by Energy Classifier with various LIMIT values.

Again, as expected, the decrease in total energy spent comes from mitigating static power consumption. On the other hand, shrinking cache size causes more conflicts, and therefore more cache misses and cache writes. This is also what causes the L4 configuration to spend more than 50 percent of baseline energy in some cases, contrary to what one might think would happen. When the LIMIT value is equal to four times the number of cores, as in L4, Energy Classifier cannot allocate more than half of the cache. The reason is that, even in the most cache-demanding scenario, where all applications are classified as harmless and are given two units of cache space, the maximum total weight which can be achieved will be equal to the half of LIMIT. However, the results show that L4 spends more than 50 percent

of baseline energy in some workloads. The difference is caused by the increase in dynamic energy consumption, as described above. This also demonstrates why having a LIMIT value greater than twice the number of cores is not a viable option.

As discussed in Chapter 4.2.1., when the simulations are run for a fixed number of cycles instead of a fixed number of instructions, measuring the total energy spent can be misleading. Therefore, analyzing the energy spent per instruction can be a more accurate way of comparing different configurations.

Figure 4.10. shows the average energy per instruction gain for all configurations compared to the baseline Set Partitioning. It can be seen from the results that the E/IPC figures follow the same trends with total energy results. Moreover, E/IPC gains are almost identical to energy gains as throughput losses for these configurations are small.

The E/IPC and efficiency results of the Energy Classifier is tightly coupled with the throughput loss. The empirical results are close to the theoretical expectations: as the LIMIT value increases, Energy Classifier starts turning off bigger portions of the cache, causing greater throughput losses. However, there are some exceptional workloads where this relationship works in the opposite direction. For example, L1 performs 5.51 percent better than Set Classifier in workload 23. The reason behind this unexpected behavior is the decay rate metric used during thread classification. The decay metric measures how many lines would decay due to inactivity if Cache Decay was applied. The number of cache decays is related to the uniformity of accesses and the availability of cache space: if a partition has a much higher number of sets than it needs, accesses would be distributed more sparsely throughout sets, increasing inactivity even if the distribution was uniform. Similarly, the non-uniformity caused by the application behavior and Fast Set Redirection (FSR) circuitry also helps accesses concentrate on some sets while leaving the others inactive.

In workload 23, the discrepancy in throughput begins at the end of the third epoch. The throughputs are identical for the first epoch due to a cold start. Both policies make the same allocation decision at the end of the first epoch as they use identical runtime statistics. At the end of the second period, throughputs of Set Classifier and L1 are similar (L1 causes a 0.5 percent loss) since cache space is under-utilized and the system can endure a reduction in cache size. However, even though the throughput is not affected greatly in the second period, the change in partition sizes skew the decay statistics in favor of less inactivity. In the second

period, L1 turns off 25 percent of the cache, causing all partitions to become 25 percent smaller than what they would be under Set Classifier. With a smaller number of sets, access distribution becomes more uniform, lowering the decay metric. While some threads are classified as Very Harmful with Set Classifier due to high decay rates, these same threads are classified as Harmless with L1 due to their low decay rates. As a result, these threads get cache space in L1 as opposed to getting no cache space in Set Classifier. This change in classification causes a significant throughput difference: L1 performs 6.72 percent better than Set Classifier in the third epoch.

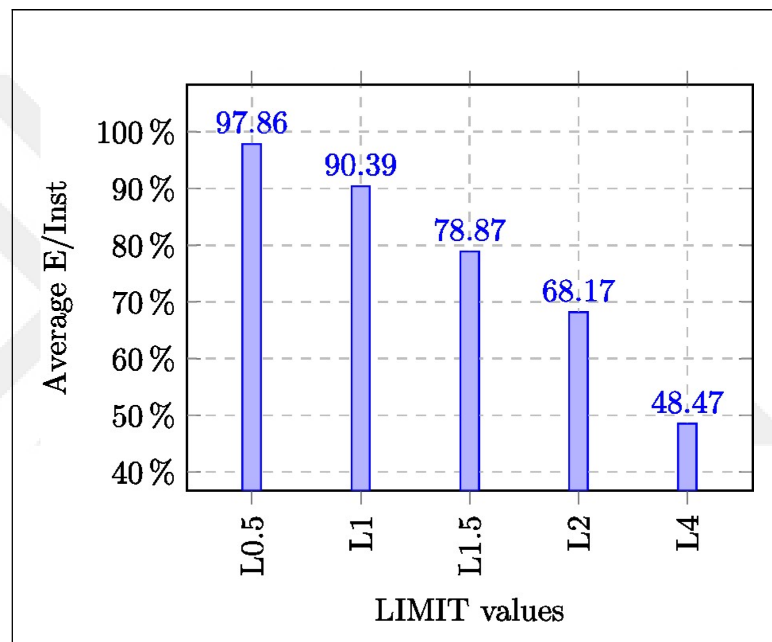


Figure 4.10. The average energy spent per instruction by Energy Classifier.

It must be noted that Energy Classifier is not an allocation policy that was designed specifically to provide high throughput and efficiency. It is merely a slightly modified version of Set Classifier, and should be taken as a proof-of-concept policy that shows Set Partitioning can be utilized as an energy-saving technique. Despite its shortcomings, Energy Classifier does not cause great throughput losses as shown in Figure 4.11.: 0.01 percent, 0.04 percent, -0.34 percent, 0.83 percent and 5.06 percent compared to baseline Set Classifier in L0.5, L1, L1.5, L2 and L4 respectively. Even when the anomalous workloads such as workload 23 are discarded throughput losses are still at reasonable levels: -0.01 percent, 0.42 percent, 0.37 percent, 1.62 percent and 6.19 percent.

The energy-saving and throughput loss results show that Energy Classifier can decrease energy consumption by 20 percent while degrading the throughput less than 1 percent with L2 configuration. Figure 4.12. shows that L2 can achieve an efficiency gain of 28 percent compared to the baseline Set Classifier. The L4 can achieve an efficiency gain of 37 percent but has a relatively high throughput loss (5 percent). The throughput loss can be traded away with the throughput gain Set Partitioning provides, turning the partitioning mechanism into one which is focused solely on energy-saving rather than improving the throughput. Still, L4 is not a feasible option since it never utilizes at least half of the cache, which does not indicate that L4 is a good configuration, but indicates that the benchmarks used for evaluation fail to generate a sufficient demand on cache resources and use the cache do not use these resources efficiently in terms of energy. However, the L2 configuration presents a good choice due to its low throughput loss and high energy savings, while never dismissing any portion of the cache indefinitely. It is also worth noting that the reported decrease in energy consumption focuses only on the cache, and further energy reduction in the processor can be achieved through a decrease in execution time.

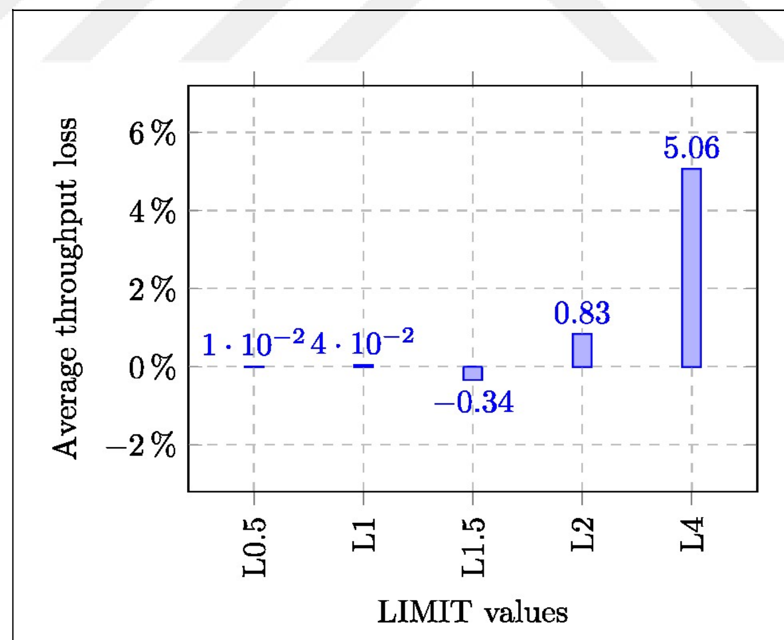


Figure 4.11. The average throughput loss caused by Energy Classifier.

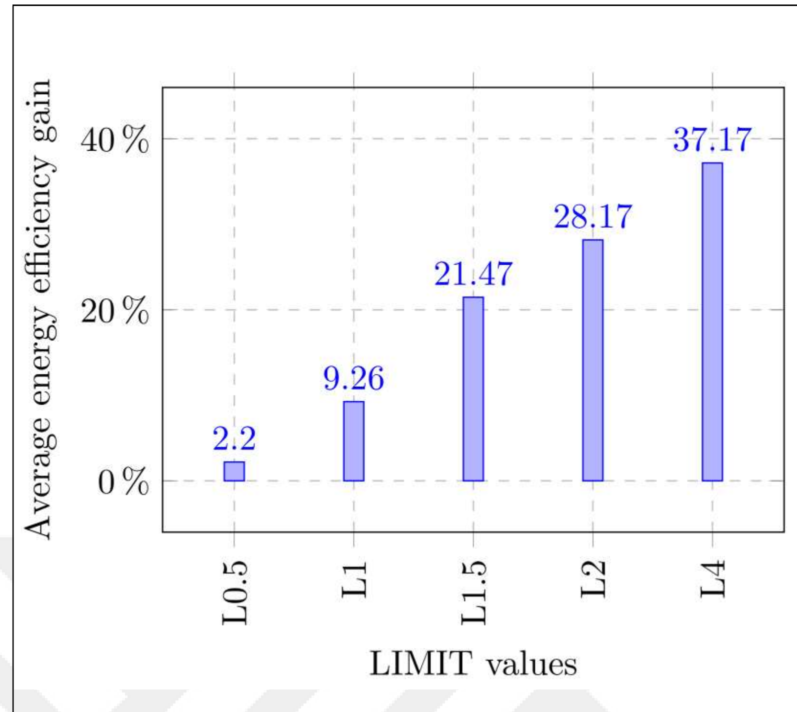


Figure 4.12. The average energy efficiency gain of Energy Classifier.

5. RELATED WORKS

Earlier work in cache partitioning focus on partitioning a shared cache among different reference streams of a single application, or among applications which run sequentially in a single processing core [24,25]. With the increased availability of multicore processors, work in this field shifted towards partitioning the cache among multiple concurrently running applications.

Brock et al. showed that the optimal partitioning-only solution is the optimal partition-sharing solution, and the problem of sharing partitions among programs can be reduced to the problem of partitioning-only [1]. Partitioning a shared cache can be done either implicitly by altering the line replacement policy [2,26-29] or explicitly by restraining the number of cache lines an application can have [3,5,18,30-36]. Explicit cache partitioning techniques require allocation policies [3,9,37-39] to determine how much cache space each partition should have. Recently, El-Sayed et al. present a partition-sharing mechanism which works on a commodity multicore system with a way partitioning scheme [40].

Most set partitioning techniques use a software approach known as page coloring [41-46], which is a technique of altering the way operating system allocates memory pages so that pages from different partitions map to different cache sets.

Our set-based partitioning scheme requires changes in how memory addresses are mapped to cache sets. Zarandi et al. propose a mechanism in which different sets have different associativity [47]. Sanchez and Kozyrakis utilize hash functions and multi-level searches to increase effective associativity [48]. Ranganathan et al. examine partitioning the cache based on sets in general terms [49]. Varadarajan et al. describe an architecture where caches consist of small, direct-mapped caching units [50].

5.1. SIMILAR WORK

Large cache structures can be examined broadly under two categories: Uniform Cache Architecture (UCA) and Non-Uniform Cache Architecture (NUCA) designs [51]. UCA caches consist of one monolithic block of cache space where all data is stored and every cache access is directed to. NUCA caches consist of several banks that can be placed

physically away from each other. NUCA caches can improve performance due to two main reasons: 1) each cache bank is smaller than the UCA block, decreasing complexity, latency and power consumption for each bank compared to the UCA design and 2) wire delays can be mitigated by placing cache lines in banks which are physically closer to their owning cores. NUCA architectures also require an interconnect circuitry so that cores can access cache banks which they are not directly connected to.

NUCA caches can be partitioned statically (SNUCA) or dynamically (DNUCA). SNUCA caches use a fixed hashing function, therefore the partition sizes can not be changed dynamically according to the workload's needs. DNUCA, on the other hand, require tag-lookups in multiple cache banks and may increase the energy consumption of the cache greatly.

5.1.1. DR-SNUCA

Gupta et al. propose an improved DNUCA architecture called DR-SNUCA which decreases energy consumption greatly while mostly attaining system throughput compared to DNUCA architectures [52]. DR-SNUCA decouples tags and data in cache lines, placing these on separate tag arrays data arrays. Least significant bits of tag field of a memory address serves as an index for the tag array. If there is a tag match, the least significant bits stored in the tag array then serve as the index of the cache line in the data arrays. This allows DR-SNUCA to place cache lines anywhere in the data array without worrying about moving the data around. Whenever the size of a partition is changed, tags that are responsible for pointing to the data's index are moved to their new location in the tag array, reducing the cache line migration problem to a simpler tag migration one. During transitions, tags are checked in both old and new locations, and the older one is invalidated to preserve the coherency.

There are some similarities between the DR-SNUCA and Set Partitioning designs, as they both partition a shared cache among many cores dynamically, in a set-based fashion. However, these designs differ from each other in several ways:

- DR-SNUCA aims to decrease the energy consumption of DNUCA caches, while Set Partitioning can be applied to both NUCA and UCA caches.

- DR-SNUCA does not partition the cache in set granularity, but it partitions the cache in cache arrays, and cache arrays are allocated to applications in powers of two. Set Partitioning can shift as small as one cache set from one partition to another.
- How the allocation policy proposed in DR-SNUCA exactly works is unclear. The optimization goal of the allocation policy is stated clearly, but how the optimal partitioning solution will be found (i.e. a greedy approach) is unclear, as well as what the complexity and cost of running the policy will be in many-core systems.
- Benchmarks are classified into three groups. One of these groups, *stream* is classified as benchmarks that do not experience a drop in miss rate when the cache size is increased. However, the results presented in the study show that when compared to the case where benchmarks are run alone, overall system throughput can drop by 80 percent when workloads which consist of only stream-type benchmarks are run.

5.1.2. Fair and Adaptive Online Set-based Cache Partitioning

The set-based cache partitioning design proposed by Abousamra et al. is the closest match to Set Partitioning mechanism proposed and examined in this research [18]. Both partitioning mechanisms work on a UCA cache, can work on set-granularity, utilize a similar redirection method and apply secondary accesses for searching the cache line in the domain of the previous partition. Set Partitioning differs from the aforementioned study in the following ways:

- Set Partitioning utilizes a hardware circuitry (FSR) which computes the redirected set index to a certain partition, whereas Abousamra et al. propose using lookup tables. FSR circuitry consumes much less energy while providing equal latency, which is discussed in Chapter 3.2.1.
- The allocation policies discussed in Set Partitioning are different from the ones proposed by Abousamra et al.
- Set Partitioning also considers using the partitioning mechanism as an energy-saving technique whereas the proposed design by Abousamra et al. focuses only on performance.
- Set Partitioning is evaluated in a many-core system, whereas the other work evaluates the design in only two- or four-core configurations. Set Partitioning is also evaluated

against another partitioning mechanism instead of being compared to the unpartitioned LRU policy.

- Submechanisms, including the ones that are common in both studies, are evaluated and discussed more thoroughly.

The work by Sahu and Ramakrishna is also close to Fair and Adaptive Online Set-based Cache Partitioning [53]. Although there are some differences between the two works such as the allocation policies, the two works are very similar in the fundamentals proposed for a set-based cache partitioning. Therefore, the differences and criticisms mentioned above also apply to Sahu and Ramakrishna's work too.

5.1.3. Jigsaw

Jigsaw is a NUCA cache management mechanism that supports data placement and partition sizing at the hardware level but leaves the decisions about these aspects to the software [9]. In addition to the proposed Share Translate Buffer, Jigsaw utilizes hashing and page table support to map data to shares. Shares are defined as partitions that consist of multiple bank partitions.

The research also proposes a novel partitioning algorithm that considers reduction both in cache misses and network latency due to multiple cache banks and cores. Jigsaw and Set Partitioning are different mechanisms in the following aspects:

- Jigsaw utilizes per-core Share Translation Buffers (STB), which operate similarly to Translation-Lookaside Buffers. Each STB has can support a small number of entries. On an STB miss, software refills the STB entries. Set Partitioning has dedicated registers for storing the beginning set and partition size for each partition.
- Jigsaw hashes the incoming address and uses the hash value to pick the array entry used. The STB translation latency is ignored as it is initiated speculatively on an L2 access. Set Partitioning implements a Fast Set Redirection circuitry for address translation, which incurs one cycle delay, and performance evaluation is done without disregarding the increase critical path delay.
- Jigsaw focuses on partitioning the cache in a NUCA configuration. The research's evaluation concludes that Jigsaw and Vantage achieve a similar reduction in overall

cache misses, the superiority of Jigsaw over Vantage is in reducing network delays. Set Partitioning does not focus on NUCA architectures, and outperforms Vantage, which provides similar performance to Jigsaw in non-NUCA caches.

5.2. VANTAGE

In this section the inner workings of the Vantage partitioning mechanism are explained in detail, as Set Partitioning is evaluated by comparing performance and fairness outputs to of Vantage [5].

Vantage is a scalable line-based partitioning mechanism that aims to allocate cache lines equal to their target allocation throughout the cache. To do this, Vantage utilizes several mechanisms. The first mechanism Vantage utilizes is a variation of a pseudo-LRU replacement policy. In this policy, each partition has two 8-bit timestamps called SetpointTS and CurrentTS. Whenever a new line is inserted into the cache, its timestamp is set to the CurrentTS of the inserting partition. Both timestamps are increased by one when the number of cache accesses reaches one-sixteenth of the lines the partition has in the cache.

Before a cache line is inserted into a set, all lines in that set are examined. If an examined cache line belongs to a partition that has more lines in the cache than allocated, and the timestamp is not between CurrentTS and SetpointTS, that line is demoted to the unmanaged partition. The unmanaged partition is a virtual partition that represents an additional partition to mark cache lines for eviction. After all lines are examined, the evicted line is selected from the unmanaged partition with the oldest timestamp value. If no unmanaged lines are found, a line is selected for eviction at random.

After 256 candidates have been examined in multiple cache accesses, Vantage consults a table to find out if the partition has demoted sufficient lines throughout these accesses. If it did, SetpointTS is decremented by one, expanding the interval between SetpointTS and CurrentTS, allowing the partition to demote fewer lines in the future. If the partition did not demote enough lines in the last 256 candidate examinations, SetpointTS is incremented by one to force the partition to demote more lines in the future. The threshold number of demotions in the table is calculated at the beginning of each repartitioning period according

to the target size of the partition and the theoretical size determined for the unmanaged partition.

Vantage aims to allocate cache space to partitions by guiding the system towards the target allocation by increasing/decreasing the likelihood of partitions evicting lines. This is achieved by contracting/expanding the interval between the two timestamps. The system requires a set amount of cache space to be allocated to the unmanaged partition. Increasing the size of the unmanaged partition improves isolation and reduces inter-partition interference, but reduces the amount of cache space which can be dedicated to partitions. In fact, the unmanaged partition acts as an additional partition that is shared among all other partitions.

Vantage partitioning fails to provide sufficient associativity at times. There may be several reasons that can cause this: the unique memory behavior of the workload mix, non-uniform distribution of cache accesses and low unmanaged partition size. Set Partitioning guarantees full associativity provided by the set-associative structure of the cache and complete isolation. Empirical results presented in Chapter 4.1.2.4. show that cache lines evicted from partitions that are under their target allocation can reach up to 75 percent of all evictions, and the workloads/benchmarks that perform better with Set Partitioning share a high rate of such eviction with Vantage.

6. CONCLUSIONS AND FUTURE WORK

As the number of real and virtual cores in a single processor increase, scalable cache partitioning methods become even more important. A set-based, scalable mechanism that can provide higher performance and fairness than Vantage is described in this research.

We start by describing the basics of a set-based partitioning mechanism, how should Set Partitioning redirect the accesses to an interval of sets dedicated to a partition and how to overcome the problem of inaccessible dirty lines. Then, we propose a simple circuit (Fast Set Redirection) that will carry out the set index computation, and propose storing access information for the previous period to initiate a secondary access that can retrieve lines which become inaccessible due to the last change in allocation decision (Double Access).

Secondly, an allocation policy (Set Classifier) that is designed with set-based partitioning in mind is proposed. The traffic (based on number of accesses), miss and decay (number of sets unused for a set amount of time) statistics are collected during execution, and Set Classifier classifies partitions into four categories based on these metrics. The four classes are Null (not interested in cache), Harmless (utilizes the cache well), Harmful (benefits less from cache space, slightly hurts other partitions) and Very Harmful (significantly harmful to the other applications). Due to the low benefit obtained Set Classifier does not allocate any cache space to these classes. Harmless partitions receive double the amount of cache space Harmful partitions do.

We then evaluate our design in a 32-core configuration in simulation. We begin by analyzing the contribution of each sub-mechanism (Fast Set Redirection, Double Access, Set Classifier and the modified set index computation) in terms of performance and fairness. We then compare Set Partitioning to Vantage [5]. Set Partitioning provides around 5 percent improvement for both performance and fairness.

We also investigate the possibility of using Set Partitionng as an energy-saving technique. Set Partitioning is abstracted as shrinking the cache size dynamically, and a semi-theoretical analysis is conducted by comparing it with similar energy-saving techniques. Then, an example allocation policy is proposed to prove the viability of utilizing Set Partitioning as an energy-saving method. With the simple modifications in Set Classifier, Set Partitioning provides a 28 percent reduction in energy spent per instruction and energy efficiency at the

cost of less than 1 percent loss in performance. The effects of Set Partitioning itself is also examined in terms of energy (Fast Set Redirection circuitry, Double Access, and the cost of flushing cache lines after repartitioning).

The importance of cache partitioning has been shown many times in the literature. Modern partitioning mechanisms should be scalable, as the number of partitions increases due to the increase in the number of cores. Another aspect which is also of great importance is protecting associativity, which is the main motivation for this research. Associativity plays a key role in cache performance and many previous schemes hurt the associativity due to the nature of the schemes themselves. Set Partitioning can provide explicit partitioning, strong isolation, high scalability while protecting associativity. Additionally, Set Partitioning can be modified to act also as an energy-saving technique.

Both the enforcement mechanism and the allocation policies described in this research are proposed as proof-of-concept designs with the purpose of investigating the viability of set-based cache partitioning as a means of improving performance, fairness and energy efficiency. Possible future work includes further improvements both in minor and major aspects of the design. These improvements include:

- Investigation of a new sub mechanism which gradually traverses sets throughout a re-partitioning period and writes back dirty stale lines. This approach can help mitigate the harmful effects of re-partitioning and allow Double Access mechanism to be limited to a certain interval at the beginning of a re-partitioning period, reducing additional energy consumed by Double Access mechanism.
- Investigation of the application of orthogonal techniques. For example, Talus doubles the number of partitions to remove cliffs in miss curves [4]. Doubling the number of partitions in a many-core configuration such as 32 cores requires a highly scalable partitioning mechanism that can provide high performance, such as Set Partitioning. Similarly, ZCache is a technique that can improve associativity beyond the number of set-associative ways inside the cache [48]. Set Partitioning is a design which does not conflict with such mechanisms, and can be adapted to support these other mechanisms to further improve system performance.
- Investigation of different allocation policies. Energy Classifier is an introduction-level policy with only simple modifications to the baseline Set Classifier. Energy

Classifier can be further improved by either implementing a dynamic LIMIT value that can adapt to the workload or by being redesigned to optimize the energy efficiency of the system.

- Investigation of new structures that can collect valuable runtime statistics. Utility Monitors can still provide useful information, but the statistics gathered from these structures have limited benefit to Set Partitioning as Utility Monitors provide hit curves for cache ways whereas Set Partitioning allocates sets [3]. A new structure that can provide similar information for sets can help improve the allocation policy greatly.
- Investigation of the utilization of Set Partitioning as a security technique. Currently, Set Partitioning can provide strict isolation and can be used to provide security against side-channel attacks with minor modifications. For example, some cores may be marked as critical cores that only applications that are selected by the OS can run on. In such a system, malignant applications can't run on the critical cores, and due to strict isolation of Set Partitioning can't access/flush lines from the cache space dedicated to these cores. Alternatively, additional partitions may be defined where only certain applications determined by the OS are allowed to use.

7. LIMITATIONS

Processor development is a long, complicated and costly process. It is not possible for researchers to always manufacture their designs as physical processors and evaluate them in a real system. Therefore, researchers tend to use software simulators that imitate real processors to some extent.

Naturally, the cost of using a simulator increases as the level of accuracy increases (cost of development, execution time). Simulators don't aim to produce results that would be exactly the same with a real processor, they rather aim to balance the tradeoff between the cost and the accuracy so that researchers can gather empirical data for evaluation in a feasible way.

This research is also subject to these conditions and utilizes a software-based simulator called Macsim to collect data [19]. Every simulator has its strengths and weaknesses, and the results presented in this research is only as good as the simulator can provide.

The empirical data and the evaluation is also dependant on the applications used as benchmarks. Diversity among benchmarks is an important factor in providing reliable results. We utilize the SPEC2006 benchmark suite based on its diversity, wide acceptance in the computer architecture community and availability to us during the research [21]. We utilize all benchmarks except the ones that produced errors when executed in the simulator (perlbench, gcc and cactusADM).

This research focuses on multi-core configurations and the number of possible workloads increases drastically in such configurations. Due to time constraints, only a small number of workloads were used for evaluation. For the same reason, the amount of simulated time for each run if the simulation is also limited.

The simulated benchmarks, and therefore workloads, produce a relatively low demand in cache memory. When there is a larger cache space available than the combined needs of all applications in the workload, the system can provide high performance even without a partitioning mechanism to regulate the cache usage. The purpose of this research is to investigate and evaluate a set-based partitioning mechanism when there is sufficient competition for the cache. The simulated cache configurations are chosen to strike a balance

between realistic configurations and small enough cache sizes to generate competition among applications.

REFERENCES

1. Brock J, Ye C, Ding C, Li Y, Wang X, Luo Y. Optimal cache partition-sharing. *Proceedings of the International Conference on Parallel Processing (ICPP)*; 2015: IEEE.

2. Xie Y, Loh GH. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. *International Symposium on Computer Architecture (ISCA)*; 2009: IEEE/ACM.

3. Qureshi MK, Patt YN. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *Proceedings of Annual International Symposium on Microarchitecture (MICRO)*; 2006: IEEE.

4. Beckmann N, Sanchez D. Talus: A simple way to remove cliffs in cache performance. *International Symposium on High Performance Computer Architecture (HPCA)*; 2015: IEEE.

5. Sanchez D, Kozyrakis C. Vantage: Scalable and efficient fine-grain cache partitioning. *Proceedings of the International Symposium on Computer Architecture (ISCA)*; 2011: IEEE/ACM.

6. Andrea A, Edler T. On global electricity usage of communication technology: Trends to 2030. *Challenges*. 2015;6(1):117-57.

7. Kaxiras S, Hu Z, Martonosi M. Cache-decay: Exploiting generational behavior to reduce cache leakage power. *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*; 2011: IEEE.

8. Flautner K, Kim NS, Martin S, Blaauw D, Mudge T. Drowsy caches: Simple techniques for reducing leakage power. *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*; 2002: IEEE.

9. Beckmann N, Sanchez D. Jigsaw: Scalable software-defined caches. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*; 2013: IEEE.

10. Qureshi MK. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. *Proceedings of Annual International Symposium on Microarchitecture (MICRO)*; 2018: IEEE.

11. Vassiliadis S, Phillips J, Blaner B. Interlock collapsing ALU's. *IEEE Transactions on Computers*. 1993;42(7):825-839.

12. Ovant BS, Güney IA, Savaş ME, Kücük G. Last level cache partitioning via multiverse thread classification. *Turkish Journal of Electrical Engineering and Computer Science*. 2018;26(1):220-233.

13. Fog A. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs; [cited 2019 1 December]. Available from: https://www.cs.utexas.edu/~hunt/class/2016-spring/cs350c/documents/Agner-Fog/instruction_tables.pdf.

14. Powell M, Yang SH, Falsafi B, Roy K, Vijaykumar TN. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. *Proceedings of the International Symposium on Lower Power Electronics and Design (ISPLED)*; 2000.

15. Powell M, Yang SH, Falsafi B, Roy K, Vijaykumar TN. Reducing leakage in a high-performance deep-submicron instruction cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. 2001;9(1):77-89.

16. Laros III JH, Pedretti K, Kelly SM, Shu W, Ferreira K, Vandyke J, Vaughan C. Energy delay product. *Energy-Efficient High Performance Computing*. 2013:51-55.

17. Balasubramonian R, Kahng AB, Muralimanohar N, Shafiee A, Srinivas V. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization*. 2017;14(2):1-25.

18. Abousamra S, El-Mahdy A, Selim S. Fair and adaptive online set-based cache partitioning. *The International Conference on Computer Engineering & Systems*; 2011: IEEE.

19. Kim H, Lee J, Lakshminarayana NB, Sim J, Lim J, Pro T. MacSim: A CPU-GPU heterogeneous simulation framework; [cited 2019 1 December]. Available from: http://comparch.gatech.edu/hparch/macsim/macsim.pdf.

20. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, et al. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices*. 2005;40(6):190-200.

21. Henning JL. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*. 2006;34(4):1-17.

22. Vandierendonck H, Seznec A. Fairness metrics for multi-threaded processors. *IEEE Computer Architecture Letters*. 2011;10(1):4-7.

23. Mattson RL, Gecsei J, Slutz DR, Traiger IL. Evaluation techniques for storage hierarchies. *IBM Systems Journal*. 1970;9(2):78-117.

24. Stone HS, Turek J, Wolf JL. Optimal partitioning of cache memory. *IEEE Transactions on Computers*. 1992;41(9):1054-1068.

25. Suh GE, Kurian G, Devadas S, Rudolph L. Analytical cache models with applications to cache partitioning. *Proceedings of the International Conference on Supercomputing (ICS)*; 2001: ACM.

26. Qureshi MK, Jaleel A, Patt YN, Steely SC, Emer J. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*. 2007;35(2):381-391.

27. Jaleel A, Hasenplaugh W, Qureshi M, Sebot J, Steely S, Emer J. Adaptive insertion policies for managing shared caches. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*; 2008: IEEE.

28. Duong N, Zhao D, Kim T, Cammarota R, Valero M, Veidenbaum AV. Improving cache management policies using dynamic reuse distances. *Proceedings of Annual International Symposium on Microarchitecture (MICRO)*; 2012: IEEE.

29. Li L, Lu J, Cheng X. Block value based insertion policy for high performance last-level caches. *Proceedings of the ACM international conference on Supercomputing (ICS)*; 2014: ACM.

30. Chiou D, Devadas S, Rudolph L, Ang BS. Dynamic cache partitioning via columnization. *Proceedings of the Annual Design Automation Conference (DAC)*; 2000.

31. Suh GE, Rudolph L, Devadas S. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*. 2004;28(1):7-26.

32. Wang X, Martinez JF. XChange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures. *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*; 2015: IEEE.

33. Wang R, Chen L. Futility Scaling: High-associativity cache partitioning. *Proceedings of Annual International Symposium on Microarchitecture (MICRO)*; 2015: IEEE.

34. Chang J, Sohi GS. Cooperative cache partitioning for chip multiprocessors. *Proceedings of the Annual International Conference on Supercomputing (ICS)*; 2007: ACM.

35. Manikantan R, Rajan K, Govindarajan R. Probabilistic shared cache management (PriSM). *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*; 2012: IEEE.

36. Sundararajan KT, Porpodas V, Jones TM, Topham NP, Franke B. Cooperative partitioning: Energy-efficient cache partitioning for high-performance CMPs. *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*; 2012: IEEE.

37. Kim S, Chandra D, Solihin Y. Fair cache sharing and partitioning in a chip multiprocessor architecture. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*; 2004: IEEE.

38. Guney IA, Yildiz A, Bayindir IU, Serdaroglu KC, Bayik U., Kucuk G. A machine learning approach for a scalable, energy-efficient utility-based cache partitioning. *Proceedings of the International Conference on High Performance Computing (HPC)*; 2015: Springer.

39. Lin X, Balasubramonian R. Refining the utility metric for utility-based cache partitioning. *Proceedings of the Workshop on Duplicating, Deconstructing, and Debunking*; 2011.

40. El-Sayed N, Mukkara A, Tsai PA, Kasture H, Ma X, Sanchez D. KPart: A hybrid cache partitioning-sharing technique for commodity multicores. *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*; 2018: IEEE.

41. Wang X, Chen S, Setter J, Martinez JF. SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support. *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*; 2017: IEEE.

42. Ye Y, West R, Cheng Z, Li Y. COLORIS: A dynamic cache partitioning system using page coloring. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*; 2014: IEEE.

43. Lin J, Lu Q, Ding X, Zhang Z, Zhang X, Sadayappan P. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*; 2008: IEEE.

44. Sherwood T, Calder B, Emer J. Reducing cache misses using hardware and software page placement. *Proceedings of the Annual International Conference on Supercomputing (ICS)*; 2008: ACM.

45. Tam D, Azimi R, Soares L, Stumm M. Managing shared L2 caches on multicore systems in software. *Workshop on the Interaction between Operating Systems and Computer Architecture*; 2007.

46. Zhang L, Liu Y, Wang R, Qian D. Lightweight dynamic partitioning for last level cache of multicore processor on real system. *Proceedings of the International Conference on Parallel and Distributed Computing (PDCAT)*; 2012: IEEE.

47. Zarandi HR, Sarbazi-Azad H. Hierarchical binary set partitioning in cache memories. *Journal of Supercomputing*. 2005;31(2):185–202.

48. Sanchez D, Kozyrakis C. The zcache: Decoupling ways and associativity. *Proceedings of Annual International Symposium on Microarchitecture (MICRO)*; 2010: IEEE.

49. Ranganathan P, Adve S, Jouppi NP. Reconfigurable caches and their application to media processing. *ACM SIGARCH Computer Architecture News*. 2000;28(2):214-24.

50. Varadarajan K, Nandy SK, Sharda V, Bharadwaj A, Iyer R, Makineni S, et al. Molecular caches: A caching structure for dynamic creation of application-specific heterogeneous

cache regions. *Proceedings of Annual International Symposium on Microarchitecture (MICRO)*; 2006: IEEE.

51. Kim C, Burger D, Keckler SW. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*; 2002.

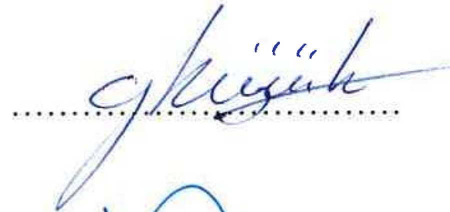
52. Gupta A, Sampson J, Taylor MB. DR-SNUCA: An energy-scalable dynamically partitioned cache. *Proceedings of the International Conference on Computer Design (ICCD)*; 2013: IEEE.

53. Sahu A, Ramakrishna S. Creating heterogeneity at run time by dynamic cache and bandwidth partitioning schemes. *Proceedings of the Annual Symposium on Applied Computing (SAC)*; 2014.

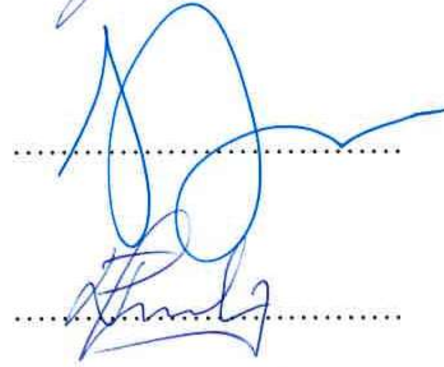
EVALUATION OF LAST LEVEL SET-BASED CACHE PARTITIONING TECHNIQUES IN TERMS OF PERFORMANCE, POWER AND FAIRNESS

APPROVED BY:

Prof. Dr. Gürhan KÜÇÜK
(Thesis Supervisor)
(Yeditepe University)



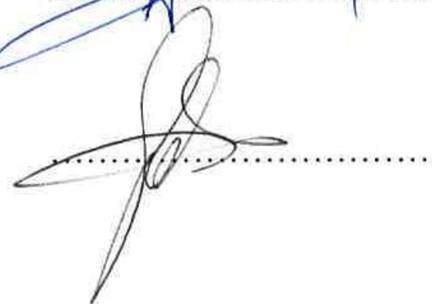
Prof. Dr. Sezer GÖREN UĞURDAĞ
(Yeditepe University)



Prof. Dr. Fatih UĞURDAĞ
(Özyeğin University)

Assist. Prof. Dr. Onur DEMİR
(Yeditepe University)

Assist. Prof. Dr. Erdinç ÖZTÜRK
(Sabancı University)



DATE OF APPROVAL: ..../..../2020